NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

Microcontrollers Applications (Smart Cards)

Student: **Mohammad Al-Ali (20011012)**

Supervisor: **Mr. Jamal Fathi**

Lefkoşa - 2005

ACKNOWLEDGMENTS

My utmost thanks to my Lord Allah that i could complete my graduation project.

I could not have prepared this project without the generous help of my family,supervisor,and friends.

First, I would like to thank my family. I could never have prepared this project without the encouragement and support of my parents, brothers, and sisters.

The root of this success lies under the most affectionate wish of my loving FATHER. I am grateful to him to assist me to grow in knowledge. I salute you, my father.

My deppest thanks to my brother Eng. Khalil Al.ali who supplied the warmth, enthusiasm, and assist to finish my education , I will never forget him, so my regards and my love to him.

I would like also to deeply thank my supervisor assoc. Mr. Jamal fathi for his invaluable advice, and belief in my work and me over all the courses of this Degree.

I would like also to express my gratitude to Near East University for that made the work possible.

I would like also to thank all my friends who were always available for my assistance throughout my studies.

ABSTRACT

This project presents the P87LPC760 is a 14-pin single chip microcontroller designed for applications demanding high integration, low cost solutions over a wide range of performance requirements. It is based on an 80C51 processor architecture that executes instructions at twice the rate of standard 80C51 devices.

The P87LPC760 offers internal RC operation, wide operating voltage range, programmable I/O port configurations, LED drive outputs, two 16 bit timers, a build-in watchdog timer, four keypad interrupt inputs and power reduction modes.

All these features make the LPC760 very suitable for remote control transmitter applications. It is a very cost effective alternative for older or even discontinued devices like the PCA84C122 and the SAA3010.

Smart cards and their operating systems. First smart cards categories, evolution, their hardware, their software and general operating systems are discussed. Then main open operating systems that have bigger market exposure such as java card, MULTOS, windows card are described in successive chapters.

CONTENTS

	AKNOWLEDGEMENTS	**i**
	ABSTRACT	**ii**
	CONTENTS	**iii**
1.	**INRODUCTION**	**1**
	1.1 Overview	1
	1.2 Numbering Systems and Code Sets	2
	1.2.1 Numbering Systems	2
	1.2.2 Code Sets	7
2.	**TYPES OF MEMORY**	**10**
	2.1 Overview	10
	2.2 Code Memory	11
	2.3 External RAM	12
	2.4 On Chip Memory	12
	2.4.1 Register Banks	14
	2.4.2 Bit Memory	15
	2.5 Special Function Register (SFR) Memory	17
	2.6 What Are SFRs?	18
	2.6.1 SFR Types	20
	2.6.2 SFR Descriptions	20
	2.6.3 Other SFRs	27
3.	**MICROCONTROLLERS APPLICATIONS**	**28**
	3.1 Introduction	28
	3.2 Hardware	28
	3.3 RC5 Transmission Protocol	29
	3.4 Software	31
	3.4.1 Main Loop	31

3.4.2 Keyboard Interrupt 32
3.4.3 Timer 0 (de-bounce timer) Interrupt 33
3.4.4 Watchdog Timer Interrupt 34
3.5 Send RC5 Code Word 35
3.6 36 KHz Modulator 35
4. MEASURING DUTY CYCLES WITH AN INTEL MCS-51 MICROCONTROLLER **37**
4.1 Introduction 37
4.2 Hardware-controlled Measurement via INT0 Pin 38
4.3 Measurement via INT0 Pin with Serial Communication 41
4.4 Duty-Cycle Measurement using Timer 2; Capture Register 42
4.5 Duty-Cycle Measurement using a Programmable Counter Array (PCA) 44
4.6 Software-Controlled Measurement 49
4.7 Understanding Averaging of Measurement Results 51
5. SMART CARDS AND THEIR OPERATING SYSTEMS 53
5.1 Introduction 53
5.2 Smart Card Overview 53
5.3 Smart Card Hardware 55
5.3.1 Memory System 56
5.3.2 Central Processing Unit 56
5.3.3 Smart Card Input/Output 57
5.4 Smart Card Software 57
5.5 Smart Card Standard 59
5.6 Smart Card Operating System 60
5.6.1 Smart Card File Systems 62
5.6.2 Application Protocol Data Units (APDUs) 63
5.7 JAVA Card 65
5.8 MULTOS 67
5.8.1 Assembler Programming Language 68
5.8.2 C Programming Language 68

5.8.3 Java Programming Language 68
5.8.4 Visual Basic Programming Language 69
5.9 Windows Card 70
5.10 Summary 71
6. CONCLUSION **72**
7. REFERENCES **73**

1. INTRODUCTION

1.1 Overview

Despite it's relatively old age, the 8051 is one of the most popular microcontrollers in use today. Many derivative microcontrollers have since been developed that are based on--and compatible with--the 8051. Thus, the ability to program an 8051 is an important skill for anyone who plans to develop products that will take advantage of microcontrollers.

The various chapters of the document will explain the 8051 step by step. The chapters are targeted at people who are attempting to learn 8051 assembly language programming. The appendices are a useful reference tool that will assist both the novice programmer as well as the experienced professional developer.

This document assumes the following:

- A general knowledge of programming.
- An understanding of decimal, hexadecimal, and binary number systems. For some background information on these number systems
- A general knowledge of hardware

That is to say, no knowledge of the 8051 is assumed--however, it is assumed you've done some amount of programming before, have a basic understanding of hardware, and a firm grasp on the three numbering systems mentioned above. The concept of converting a number from deciminal to hexidecimal and/or to binary is not within the scope of this document--and if you can't do those types of conversions there are probably some concepts that will not be completely understandable.

This document attempts to address the need of the *typical* programmer. For example, there are certain features that are nifty and in some cases very useful--but 95% of the programmers will never use these features.

1.2 Numbering Systems and Code Sets

1.2.1 Numbering Systems

A numbering system is a set of digits used for mathematical operations such as counting, adding, subtracting, dividing and multiplying. The numbering system that we are all familiar with is called decimal. Decimal is called a *base 10* numbering system because it uses 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). Let's now go (way) back to basics and look at a decimal number 182. From our elementary math schooling, each digit in a decimal number is in a particular column. This "column placement" as we shall call it is fundamental to any numbering system whether it is base 10 (decimal) or something else. The 2 is in the ones column, the 8 in the tens column and the 1 in the hundreds column. We could break the number out as follows:

- 2 x 1 = 2
- 8 x 10 = 80
- 1 x 100 = 100

which would give us:

- 2 + 80 + 100 = 182

Another way of determining the values for each column is by using exponents of the base numbering system. For example, 10 to the 0 power equals 1, the 1's column. 10 to the power of 1 equals 10, the 10's column, 10 to the power of 2 equals 100, the 100's column and so on. Be sure that you understand this basic math methodology to column placement before moving on.

Now when we count in decimal, we generally don't start with zero because it is implied, but in this case we will show it for completeness. So we count then as follows:

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11

and so on.

Now notice that when we got to 9, our next number had to use the next column to the left, the 10s column (10 to the power of 1). When starting a new column, the first number we start with is always 1 and the first number to the right of the 1 will always be a zero. This may seem obvious at first, especially with our most familiar numbering system decimal, but keep this concept in mind when working in other numbering systems as we will be shortly.

You're probably thinking, "Gee this is just wonderful, what else do you have for us Einstein?" Well, let us now consider a different numbering system. One that is fundamental to all computer technology. This numbering system in question is called Binary. Binary is a *base 2* numbering system which means it uses only 2 digits (zero and one).

Let's see how one might count using this numbering system:

- 0
- 1
- 10
- 11
- 100
- 101
- 110
- 111
- 1000
- 1001
- 1010
- 1011

and so on.

It may seem strange at first, even tedious, but this is how to count in binary. The concept of column placement we demonstrated using the decimal number 182 remains with binary numbers. Let's use an example binary number like 10110110. This probably means nothing to you at first glance, but if we dissect it using our knowledge of column placement, we put this binary number into a more meaningful context.

Let's first start by figuring out the values of each column for this 8 digit binary number. Working from right to the left, the first column is the 1's column (in any numbering system, the rightmost column is always the 1's column). Now how do we figure out the values for the remaining columns? Answer this question, how many digits are we using in the binary numbering system? Two. Earlier we learned that column placement values can be known by using exponents of the base number. 2 to the power of zero is one or the 1's column which we already knew (any number to the power of zero is always 1). 2 to the power of 1 is 2 (any number to the number of 1 is that number). 2 to the power of 2 is 4 (2 x 2 = 4). 2 to the power

of 3 is 8 (2 x 2 x 2 = 8). We will eventually end up with the following column values, from largest to smallest: 1's, 2's, 4's, 8's, 16's, 32's, 64's, and 128's.

Now we can use some math to figure out what the binary number equals in decimal form. Although a technique which probably seemed silly before, let's multiply each number in each column by it's column placement value:

- 0 x 1 = 0
- 1 x 2 = 2
- 1 x 4 = 4
- 0 x 8 = 0
- 1 x 16 = 16
- 1 x 32 = 32
- 0 x 64 = 0
- 1 x 128 = 128

which would give us:

- 0 + 2 + 4 + 0 + 16 + 32 + 0 + 128 = 182

Cool! You've just done your first binary to decimal conversion, congratulations! Using these techniques, you could convert any numbering system to decimal. This would certainly come in handy if you ever meet an extra terrestrial who has 3 fingers and thus uses a *base 3* numbering system. A more practical application would be to use these techniques on another numbering system that is widely in use in the computer world. This system is hexadecimal.

The hexadecimal numbering system is a *base 16* numbering system. Some people refer to this numbering system simply as "hex". The 16 digits used in hex in order from smallest to largest are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Now what are those letters doing in there?! Simple, we don't have any single digit past 9 so we just use the first 6 letters of the alphabet as digits. This means that A is 10 in decimal and F is equal to 15 in decimal. If you're still awake, perhaps you can figure out what B6 in hexadecimal equals to in decimal?

If you said 182, good guess. If you actually used our previous techniques to figure it out by hand, then you obviously are becoming a master of numbering systems. For those who need a little guidance, here is the math:

- 6 x 1 = 6
- B x 16 = 176

which would give us:

- 6 + 176 = 182

Again, the rightmost column is the 1's column, and the value for the next column is the base number to the power of 1, or simply the base number itself, in this case 16. B is really 11 in decimal, which is why B x 16 equals 176. The rest is basic (decimal) math from grade school days.

You may never count in binary, convert a hex number to decimal or meet E.T., but by understanding these concepts you can begin to understand how computers and in turn networks really work at a very fundamental level.

Before moving on to some actual data communications, let's first ask ourselves why we use binary and hex numbering systems with computers at all? What's wrong with the decimal numbering system? The problem lies in how a computer operates at the physical level. Working with electricity and electrical current, computers can represent two *states* and two states only at their most basic level. Think of the standard light bulb that is either on or off. Power is either causing the light bulb to shine (on) or the lack of power means darkness (off). Computers work with these on and off's of electricity, two states that we represent using ones and zeroes.

Hex is actually a numbering system that computers know nothing about. Hex was created and used as an easier representation for programmers. You may think there's nothing easy about a number such as F8A2 but it is a little easier to work with than all the 1's and zero's it

represents. Hex is also convenient because one hex digit represents 4 binary digits (bits). In the world of computers where we talk in bytes (which commonly refers to a group of 8 bits) at a time, 2 hex characters can easily represent one byte. In the next section on code sets this concept will hopefully make more sense in case you're a little confused.

1.2.2 Code Sets

With computers using the binary numbering system for representing data, we need to find a "code set" that can correlate an alphabet, numbering system and character set to the computer's ones and zeroes. Chances are you have heard of one popular code set that was been in use for over 100 years. Here's a hint, dots and dashes. That's right, Morse code. Before there were telephones, people could send a "wire" to another city by way of a Morse code operator. Morse code operators send signals to the other end using a binary system. Instead of ones and zeroes, they used dots and dashes. A dot was represented with a quick tone while a dash was a longer tone.

The Morse code set was a scheme that matched a certain combination of dots and dashes to the alphabet, numbers zero thru nine and some special characters. With computers, we use a similar system of code sets not unlike Morse code. One of the most popular code sets is called ASCII (American Standard Code for Information Interchange). The ASCII code set uses a combination of 8 binary digits (bits) to represent the English alphabet, the ten digits of the decimal numbering system and a number of special characters. Originally ASCII was defined as a 7 bit code set, but later expanded and sometimes referred to as Extended ASCII. With 8 bits (a combination of 8 ones and zeroes) you can come up with 256 unique combinations to represent all kinds of characters we humans might want the computer to represent.

In figure 1.1 below you will find a table of the extended ASCII code set. A brief explanation of the column and row headings is necessary. Along with the human readable ASCII characters in the table, you are given the values for the characters as they would be represented in binary, hexadecimal and decimal. The binary numbers along the top row represent the 4 *low*

order bits for the characters below. The first column of binary numbers along the left are the 4 *high order* bits. To get the complete 8 bit number for the ASCII character, you put the lower order bits onto the end of the high order bits. For example, find the lower case letter 'k' in set. Go all the way over to the left of the 'k' and write down the 4 binary numbers for it's row. Now go all the way up from 'k' and write the binary digits for it's column. If you place the first 4 bits in front of the second set of 4 bits you wrote down, you should end up with a binary number of '01101001'. That binary number is the combination of bits a computer uses to represent the lower case 'k'. At least when the computer knows to use the ASCII code set.

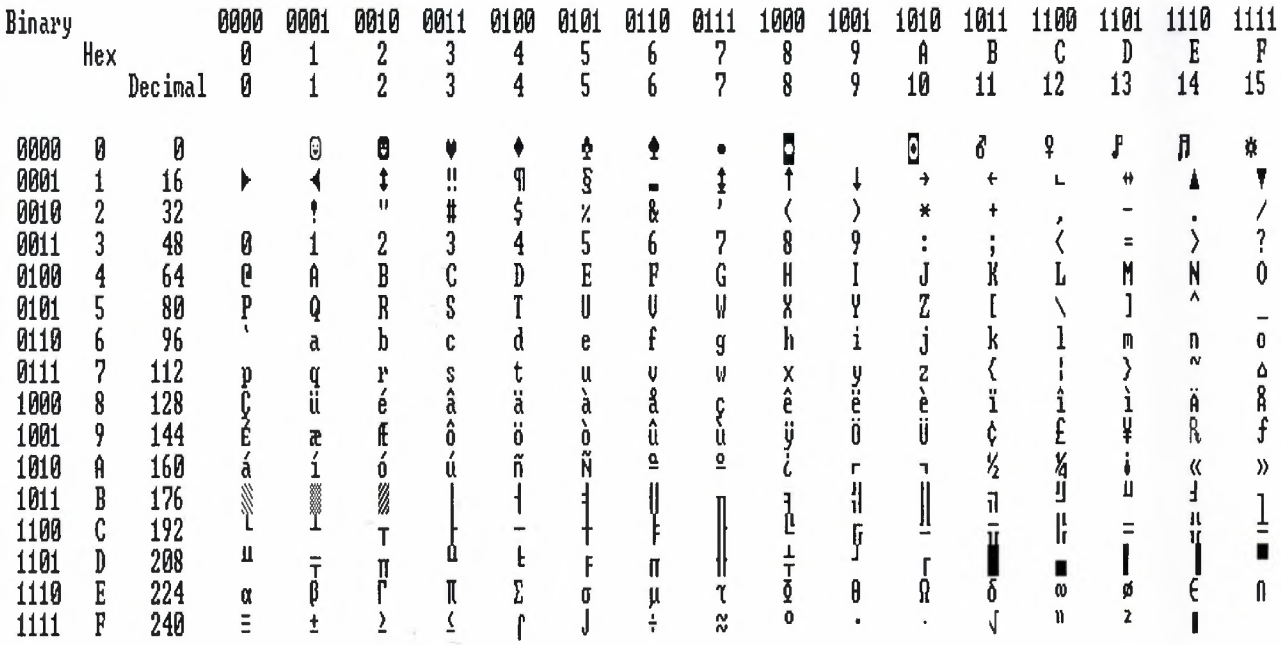


Figure 1.1 Extended ASCII Code Set

Figuring out what the hex number for each character in the table is can be found in a similar fashion. However, determining the decimal equivalent for each character is a little different. You have to add the column's decimal number to the row's decimal number. Let's do a quick test. What are the binary, hex and decimal numbers that are associated with the ASCII

character '#'? If you said 00100011, 23 and 35, you are correct. One last tip, people usually pronounce ASCII as "ASK-ee".

There are other code sets you may come across. One that's used primarily on IBM mainframes is EBCDIC (Extended Binary Coded Data Interchange Code). I think the person(s) who came up with that acronym went to the Redundancy School of Redundancy! The EBCDIC code set uses a different combination of 8 bits to represent various characters. It is not compatible with ASCII. It is usually pronounced "EB-such-dick".

2. TYPES OF MEMORY

2.1 Overview

The 8051 has three very general types of memory. To effectively program the 8051 it is necessary to have a basic understanding of these memory types.

The memory types are illustrated in figure 2.1. They are: On-Chip Memory, External Code Memory, and External RAM.

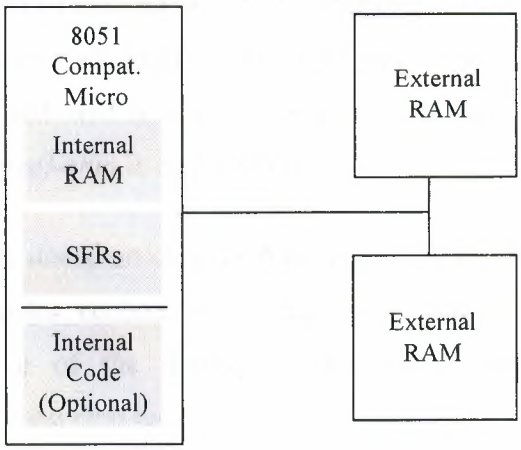


Figure 2.1 Memory Types

a) On-Chip Memory refers to any memory (Code, RAM, or other) that physically exists on the microcontroller itself. On-chip memory can be of several types, but we'll get into that shortly.

b) External Code Memory is code (or program) memory that resides off-chip. This is often in the form of an external EPROM.

c) External RAM is RAM memory that resides off-chip. This is often in the form of standard static RAM or flash RAM.

2.2 Code Memory

Code memory is the memory that holds the actual 8051 program that is to be run. This memory is limited to 64K and comes in many shapes and sizes: Code memory may be found *on-chip*, either burned into the microcontroller as ROM or EPROM. Code may also be stored completely *off-chip* in an external ROM or, more commonly, an external EPROM. Flash RAM is also another popular method of storing a program. Various combinations of these memory types may also be used--that is to say, it is possible to have 4K of code memory *on-chip* and 64k of code memory *off-chip* in an EPROM.

When the program is stored on-chip the 64K maximum is often reduced to 4k, 8k, or 16k. This varies depending on the version of the chip that is being used. Each version offers specific capabilities and one of the distinguishing factors from chip to chip is how much ROM/EPROM space the chip has.

However, code memory is most commonly implemented as off-chip EPROM. This is especially true in low-cost development systems and in systems developed by students.

Programming Tip: Since code memory is restricted to 64K, 8051 programs are limited to 64K. Some assemblers and compilers offer ways to get around this limit when used with specially wired hardware. However, without such special compilers and hardware, programs are limited to 64K.

2.3 External RAM

As an obvious opposite of *Internal RAM*, the 8051 also supports what is called *External RAM*.

As the name suggests, External RAM is any random access memory which is found *off-chip*. Since the memory is off-chip it is not as flexible in terms of accessing, and is also slower. For example, to increment an Internal RAM location by 1 requires only 1 instruction and 1 instruction cycle. To increment a 1-byte value stored in External RAM requires 4 instructions and 7 instruction cycles. In this case, external memory is 7 times slower!

What External RAM loses in speed and flexibility it gains in quantity. While Internal RAM is limited to 128 bytes (256 bytes with an 8052), the 8051 supports External RAM up to 64K.

Programming Tip: The 8051 may only address 64k of RAM. To expand RAM beyond this limit requires programming and hardware tricks. You may have to do this "by hand" since many compilers and assemblers, while providing support for programs in excess of 64k, do not support more than 64k of RAM. This is rather strange since it has been my experience that programs can usually fit in 64k but often RAM is what is lacking. Thus if you need more than 64k of RAM, check to see if your compiler supports it-- but if it doesn't, be prepared to do it by hand.

2.4 On-Chip Memory

As mentioned at the beginning of this chapter, the 8051 includes a certain amount of on-chip memory. On-chip memory is really one of two types: Internal RAM and Special Function Register (SFR) memory. The layout of the 8051's internal memory is presented in the memory map as shown in figure 2.2.

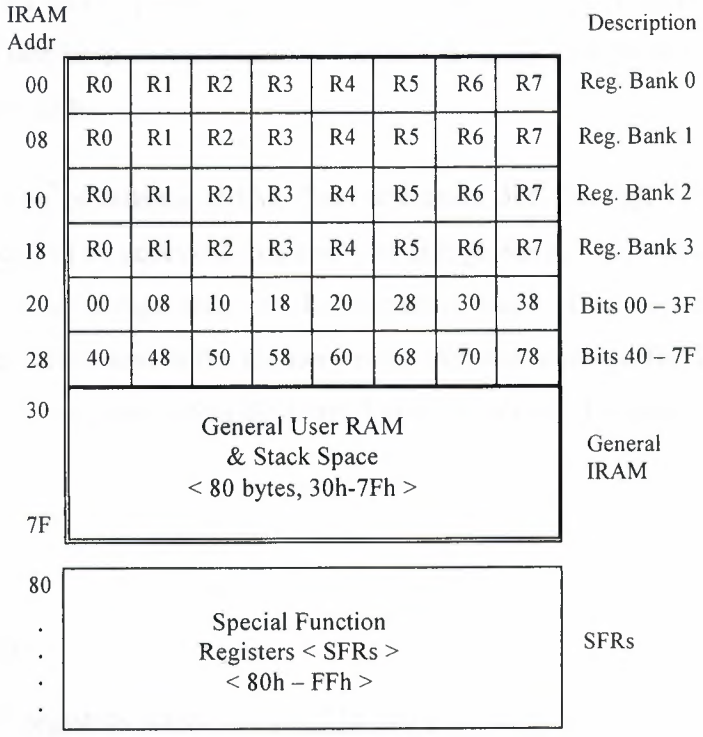


Figure 2.2 Memory Map (On-Chip Memory)

As is illustrated in figure 2.2, the 8051 has a bank of 128 bytes of *Internal RAM*. This Internal RAM is found *on-chip* on the 8051 so it is the fastest RAM available, and it is also the most flexible in terms of reading, writing, and modifying it's contents. Internal RAM is volatile, so when the 8051 is reset this memory is cleared.

The 128 bytes of internal ram is subdivided as shown on the memory map. The first 8 bytes (00h - 07h) are "register bank 0". By manipulating certain SFRs, a program may choose to use register banks 1, 2, or 3. These alternative register banks are located in internal RAM in addresses 08h through 1Fh. We'll discuss "register banks" more in a later chapter. For now it is sufficient to know that they "live" and are part of internal RAM.

Bit Memory also lives and is part of internal RAM. We'll talk more about bit memory very shortly, but for now just keep in mind that bit memory actually resides in internal RAM, from addresses 20h through 2Fh.

The 80 bytes remaining of Internal RAM, from addresses 30h through 7Fh, may be used by user variables that need to be accessed frequently or at high-speed. This area is also utilized by the microcontroller as a storage area for the operating *stack*. This fact severely limits the 8051's stack since, as illustrated in the memory map, the area reserved for the stack is only 80 bytes--and usually it is less since this 80 bytes has to be shared between the stack and user variables.

2.4.1 Register Banks

The 8051 uses 8 "R" registers which are used in many of its instructions. These "R" registers are numbered from 0 through 7 (R0, R1, R2, R3, R4, R5, R6, and R7). These registers are generally used to assist in manipulating values and moving data from one memory location to another. For example, to add the value of R4 to the Accumulator, we would execute the following instruction:

a) ADD A, R4

Thus if the Accumulator (A) contained the value 6 and R4 contained the value 3, the Accumulator would contain the value 9 after this instruction was executed.

However, as the memory map shows, the "R" Register R4 is really part of Internal RAM. Specifically, R4 is address 04h. This can be see in the bright green section of the memory map. Thus the above instruction accomplishes the same thing as the following operation:

b) ADD A, 04h

This instruction adds the value found in Internal RAM address 04h to the value of the Accumulator, leaving the result in the Accumulator. Since R4 is really Internal RAM 04h, the above instruction effectively accomplished the same thing.

But watch out! As the memory map shows, the 8051 has four distinct register banks. When the 8051 is first booted up, register bank 0 (addresses 00h through 07h) is used by default. However, your program may instruct the 8051 to use one of the alternate register banks; i.e., register banks 1, 2, or 3. In this case, R4 will no longer be the same as Internal RAM address 04h. For example, if your program instructs the 8051 to use register bank 3, "R" register R4 will now be synonymous with Internal RAM address 1Ch.

The concept of register banks adds a great level of flexibility to the 8051, especially when dealing with interrupts (we'll talk about interrupts later). However, always remember that the register banks really reside in the first 32 bytes of Internal RAM.

Programming Tip: If you only use the first register bank (i.e. bank 0), you may use Internal RAM locations 08h through 1Fh for your own use. But if you plan to use register banks 1, 2, or 3, be very careful about using addresses below 20h as you may end up overwriting the value of your "R" registers!

2.4.2 Bit Memory

The 8051, being a communications-oriented microcontroller, gives the user the ability to access a number of *bit variables*. These variables may be either 1 or 0.

There are 128 bit variables available to the user, numbered 00h through 7Fh. The user may make use of these variables with commands such as SETB and CLR. For example, to set bit number 24 (hex) to 1 you would execute the instruction:

a) SETB 24h

It is important to note that Bit Memory is really a part of Internal RAM. In fact, the 128 bit variables occupy the 16 bytes of Internal RAM from 20h through 2Fh. Thus, if you write the value FFh to Internal RAM address 20h you've effectively set bits 00h through 07h. That is to say that:

MOV 20h, #0FFh

is equivalent to:

SETB	00h
SETB	01h
SETB	02h
SETB	03h
SETB	04h
SETB	05h
SETB	06h
SETB	07h

As illustrated above, bit memory isn't really a new type of memory. It's really just a subset of Internal RAM. But since the 8051 provides special instructions to access these 16 bytes of memory on a bit by bit basis it is useful to think of it as a separate type of memory. However, always keep in mind that it is just a subset of Internal RAM--and that operations performed on Internal RAM can change the values of the bit variables.

Programming Tip: If your program does not use bit variables, you may use Internal RAM locations 20h through 2Fh for your own use. But if you plan to use bit variables, be very careful about using addresses from 20h through 2Fh as you may end up overwriting the value of your bits!

Bit variables 00h through 7Fh are for user-defined functions in their programs. However, bit variables 80h and above are actually used to access certain SFRs on a bit-by-bit basis. For

example, if output lines P0.0 through P0.7 are all clear (0) and you want to turn on the P0.0 output line you may either execute:

MOV P0,#01h

or you may execute:

SETB 80h

Both these instructions accomplish the same thing. However, using the SETB command will turn on the P0.0 line without affecting the status of any of the other P0 output lines. The MOV command effectively turns off all the other output lines which, in some cases, may not be acceptable.

Programming Tip: By default, the 8051 initializes the *Stack Pointer* (SP) to 07h when the microcontroller is booted. This means that the stack will start at address 08h and expand upwards. If you will be using the alternate register banks (banks 1, 2 or 3) you must initialize the stack pointer to an address above the highest register bank you will be using, otherwise the stack will overwrite your alternate register banks. Similarly, if you will be using bit variables it is usually a good idea to initialize the stack pointer to some value greater than 2Fh to guarantee that your bit variables are protected from the stack.

2.5 Special Function Register (SFR) Memory

Special Function Registers (SFRs) are areas of memory that control specific functionality of the 8051 processor. For example, four SFRs permit access to the 8051's 32 input/output lines. Another SFR allows a program to read or write to the 8051's serial port. Other SFRs allow the user to set the serial baud rate, control and access timers, and configure the 8051's interrupt system.

When programming, SFRs have the illusion of being Internal Memory. For example, if you want to write the value "1" to Internal RAM location 50 hex you would execute the instruction:

MOV 50h, #01h

Similarly, if you want to write the value "1" to the 8051's serial port you would write this value to the **SBUF** SFR, which has an SFR address of 99 Hex. Thus, to write the value "1" to the serial port you would execute the instruction:

MOV 99h, #01h

As you can see, it appears that the SFR is part of Internal Memory. This is not the case. When using this method of memory access (it's called direct address), any instruction that has an address of 00h through 7Fh refers to an Internal RAM memory address; any instruction with an address of 80h through FFh refers to an SFR control register.

Programming Tip: SFRs are used to control the way the 8051 functions. Each SFR has a specific purpose and format which will be discussed later. Not all addresses above 80h are assigned to SFRs. However, this area may NOT be used as additional RAM memory even if a given address has not been assigned to an SFR.

2.6 What Are SFRs?

The 8051 is a flexible microcontroller with a relatively large number of modes of operations. Your program may inspect and/or change the operating mode of the 8051 by manipulating the values of the 8051's Special Function Registers (SFRs).

SFRs are accessed as if they were normal Internal RAM. The only difference is that Internal RAM is from address 00h through 7Fh whereas SFR registers exist in the address range of 80h through FFh. Each SFR has an address (80h through FFh) and a name. Figure 2.3 provides a graphical presentation of the 8051's SFRs, their names, and their address.

80	P0	SP	DPL	DPH				PCON	87
88	TCON	TMOD	TL0	TL1	TH0	TH1			8F
90	P1								97
98	SCON	SBUF							9F
A0	P2								A7
A8	IE								AF
B0	P3								B7
B8	IP								B9
C0									C7
C8									CF
D0	PSW								D7
D8									DF
E0	ACC								E7
E8									EF
F0	B								F7

Figure 2.3 Graphical Presentations of the 8051's SFRs

As you can see, although the address range of 80h through FFh offers 128 possible addresses, there are only 21 SFRs in a standard 8051. All other addresses in the SFR range (80h through FFh) are considered invalid. Writing to or reading from these registers may produce undefined values or behavior.

Programming Tip: It is recommended that you not read or write to SFR addresses that have not been assigned to an SFR. Doing so may provoke undefined behavior and may cause your program to be incompatible with other 8051-derivatives that use the given SFR for some other purpose.

2.6.1 SFR Types

As mentioned in the chart itself, the SFRs that have a blue background are SFRs related to the I/O ports. The 8051 has four I/O ports of 8 bits, for a total of 32 I/O lines. Whether a given I/O line is high or low and the value read from the line are controlled by the SFRs in green.

The SFRs with yellow backgrounds are SFRs which in some way control the operation or the configuration of some aspect of the 8051. For example, **TCON** controls the timers, SCON controls the serial port.

The remaining SFRs, with green backgrounds, are "other SFRs." These SFRs can be thought of as auxiliary SFRs in the sense that they don't directly configure the 8051 but obviously the 8051 cannot operate without them. For example, once the serial port has been configured using **SCON**, the program may read or write to the serial port using the **SBUF** register.

Programming Tip: The SFRs whose names appear in red in the chart above are SFRs that may be accessed via bit operations (i.e., using the **SETB** and **CLR** instructions). The other SFRs cannot be accessed using bit operations. As you can see, all SFRs that whose addresses are divisible by 8 can be accessed with bit operations.

2.6.2 SFR Descriptions

This section will endeavor to quickly overview each of the standard SFRs found in the above SFR chart map. It is not the intention of this section to fully explain the functionality of each SFR--this information will be covered in separate chapters of the tutorial. This section is to just give you a general idea of what each SFR does.

a) P0 (Port 0, Address 80h, Bit-Addressable)

This is input/output port 0. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 0 is pin P0.0, bit 7 is pin P0.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

Programming Tip: While the 8051 has four I/O port (P0, P1, P2, and P3), if your hardware uses external RAM or external code memory (i.e., your program is stored in an external ROM or EPROM chip or if you are using external RAM chips) you may not use P0 or P2. This is because the 8051 uses ports P0 and P2 to address the external memory. Thus if you are using external RAM or code memory you may only use ports P1 and P3 for your own use.

b) SP (Stack Pointer, Address 81h)

This is the stack pointer of the microcontroller. This SFR indicates where the next value to be taken from the stack will be read from in Internal RAM. If you push a value onto the stack, the value will be written to the address of SP + 1. That is to say, if SP holds the value 07h, a PUSH instruction will push the value onto the stack at address 08h. This SFR is modified by all instructions which modify the stack, such as PUSH, POP, LCALL, RET, RETI, and whenever interrupts are provoked by the microcontroller.

Programming Tip: The SP SFR, on startup, is initialized to 07h. This means the stack will start at 08h and start expanding upward in internal RAM. Since alternate register banks 1, 2, and 3 as well as the user bit variables occupy internal RAM from addresses 08h through 2Fh, it is necessary to initialize SP in your program to some other value if you will be using the alternate register banks and/or bit memory. It's not a bad idea to initialize SP to 2Fh as the first instruction of every one of your programs unless you are 100% sure you will not be using the register banks and bit variables.

c) DPL/DPH (Data Pointer Low/High, Addresses 82h/83h)

The SFRs DPL and DPH work together to represent a 16-bit value called the *Data Pointer*. The data pointer is used in operations regarding external RAM and some instructions involving code memory. Since it is an unsigned two-byte integer value, it can represent values from 0000h to FFFFh (0 through 65,535 decimal).

Programming Tip: DPTR is really DPH and DPL taken together as a 16-bit value. In reality, you almost always have to deal with DPTR one byte at a time. For example, to push DPTR onto the stack you must first push DPL and then DPH. You can't simply plush DPTR onto the stack. Additionally, there is an instruction to "increment DPTR." When you execute this instruction, the two bytes are operated upon as a 16-bit value. However, there is no instruction that decrements DPTR. If you wish to decrement the value of DPTR, you must write your own code to do so.

d) PCON (Power Control, Addresses 87h)

The Power Control SFR is used to control the 8051's power control modes. Certain operation modes of the 8051 allow the 8051 to go into a type of "sleep" mode which requires much less power. These modes of operation are controlled through PCON. Additionally, one of the bits in PCON is used to double the effective baud rate of the 8051's serial port.

e) TCON (Timer Control, Addresses 88h, Bit-Addressable)

The Timer Control SFR is used to configure and modify the way in which the 8051's two timers operate. This SFR controls whether each of the two timers is running or stopped and contains a flag to indicate that each timer has overflowed. Additionally, some non-timer related bits are located in the TCON SFR. These bits are used to configure the way in which the external interrupts are activated and also contain the external interrupt flags which are set when an external interrupt has occurred.

f) TMOD (Timer Mode, Addresses 89h)

The Timer Mode SFR is used to configure the mode of operation of each of the two timers. Using this SFR your program may configure each timer to be a 16-bit timer, an 8-bit auto-reload timer, a 13-bit timer, or two separate timers. Additionally, you may configure the timers to only count when an external pin is activated or to count "events" that are indicated on an external pin.

g) TL0/TH0 (Timer 0 Low/High, Addresses 8Ah/8Ch)

These two SFRs, taken together, represent timer 0. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value.

h) TL1/TH1 (Timer 1 Low/High, Addresses 8Bh/8Dh)

These two SFRs, taken together, represent timer 1. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value.

i) P1 (Port 1, Address 90h, Bit-Addressable)

This is input/output port 1. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 1 is pin P1.0, bit 7 is pin P1.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

j) SCON (Serial Control, Addresses 98h, Bit-Addressable)

The Serial Control SFR is used to configure the behavior of the 8051's on-board serial port. This SFR controls the baud rate of the serial port, whether the serial port is activated to receive data, and also contains flags that are set when a byte is successfully sent or received.

Programming Tip: To use the 8051's on-board serial port, it is generally necessary to initialize the following SFRs: SCON, TCON, and TMOD. This is because SCON controls the serial port. However, in most cases the program will wish to use one of the timers to establish the serial port's baud rate. In this case, it is necessary to configure timer 1 by initializing TCON and TMOD.

k) SBUF (Serial Control, Addresses 99h)

The Serial Buffer SFR is used to send and receive data via the on-board serial port. Any value written to SBUF will be sent out the serial port's TXD pin. Likewise, any value which the 8051 receives via the serial port's RXD pin will be delivered to the user program via SBUF. In other words, SBUF serves as the output port when written to and as an input port when read from.

l) P2 (Port 2, Address A0h, Bit-Addressable)

This is input/output port 2. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 2 is pin P2.0, bit 7 is pin P2.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

Programming Tip: While the 8051 has four I/O port (P0, P1, P2, and P3), if your hardware uses external RAM or external code memory (i.e., your program is stored in an external ROM or EPROM chip or if you are using external RAM chips) you may not use P0 or P2. This is because the 8051 uses ports P0 and P2 to address the external memory. Thus if you are using external RAM or code memory you may only use ports P1 and P3 for your own use.

m) IE (Interrupt Enable, Addresses A8h)

The Interrupt Enable SFR is used to enable and disable specific interrupts. The low 7 bits of the SFR are used to enable/disable the specific interrupts, where as the highest bit is used to enable or disable ALL interrupts. Thus, if the high bit of IE is 0 all interrupts are disabled regardless of whether an individual interrupt is enabled by setting a lower bit.

n) P3 (Port 3, Address B0h, Bit-Addressable)

This is input/output port 3. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 3 is pin P3.0, bit 7 is pin P3.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

o) IP (Interrupt Priority, Addresses B8h, Bit-Addressable)

The Interrupt Priority SFR is used to specify the relative priority of each interrupt. On the 8051, an interrupt may either be of low (0) priority or high (1) priority. An interrupt may only interrupt interrupts of lower priority. For example, if we configure the 8051 so that all interrupts are of low priority except the serial interrupt, the serial interrupt will always be able to interrupt the system, even if another interrupt is currently executing. However, if a serial interrupt is executing no other interrupt will be able to interrupt the serial interrupt routine since the serial interrupt routine has the highest priority.

p) PSW (Program Status Word, Addresses D0h, Bit-Addressable)

The Program Status Word is used to store a number of important bits that are set and cleared by 8051 instructions. The PSW SFR contains the carry flag, the auxiliary carry flag, the overflow flag, and the parity flag. Additionally, the PSW register contains the register bank select flags which are used to select which of the "R" register banks are currently selected.

Programming Tip: If you write an interrupt handler routine, it is a very good idea to *always* save the PSW SFR on the stack and restore it when your interrupt is complete. Many 8051 instructions modify the bits of PSW. If your interrupt routine does not guarantee that PSW is the same upon exit as it was upon entry, your program is bound to behave rather erradically and unpredictably--and it will be tricky to debug since the behavior will tend not to make any sense.

q) ACC (Accumulator, Addresses E0h, Bit-Addressable)

The Accumulator is one of the most-used SFRs on the 8051 since it is involved in so many instructions. The Accumulator resides as an SFR at E0h, which means the instruction **MOV A,#20h** is really the same as **MOV E0h,#20h**. However, it is a good idea to use the first method since it only requires two bytes whereas the second option requires three bytes.

r) B (B Register, Addresses F0h, Bit-Addressable)

The "B" register is used in two instructions: the multiply and divide operations. The B register is also commonly used by programmers as an auxiliary register to temporarily store values.

2.6.3 Other SFRs

The chart above is a summary of all the SFRs that exist in a standard 8051. All derivative microcontrollers of the 8051 must support these basic SFRs in order to maintain compatibility with the underlying MSCS51 standard.

A common practice when semiconductor firms wish to develop a new 8051 derivative is to add additional SFRs to support new functions that exist in the new chip.

For example, the Dallas Semiconductor DS80C320 is upwards compatible with the 8051. This means that any program that runs on a standard 8051 should run without modification on the DS80C320. This means that all the SFRs defined above also apply to the Dallas component.

However, since the DS80C320 provides many new features that the standard 8051 does not, there must be some way to control and configure these new features. This is accomplished by adding additional SFRs to those listed here. For example, since the DS80C320 supports two serial ports (as opposed to just one on the 8051), the SFRs SBUF2 and SCON2 have been added. In addition to all the SFRs listed above, the DS80C320 also recognizes these two new SFRs as valid and uses their values to determine the mode of operation of the secondary serial port. Obviously, these new SFRs have been assigned to SFR addresses that were unused in the original 8051. In this manner, new 8051 derivative chips may be developed which will run existing 8051 programs.

Programming Tip: If you write a program that utilizes new SFRs that are specific to a given derivative chip and not included in the above SFR list, your program will not run properly on a standard 8051 where that SFR does not exist. Thus, only use non-standard SFRs if you are sure that your program will only have to run on that specific microcontroller. Likewise, if you write code that uses non-standard SFRs and subsequently share it with a third-party, be sure to let that party know that your code is using non-standard SFRs to save them the headache of realizing that due to strange behavior at run-time.

3. MICROCONTROLLERS APPLICATIONS

3.1 Introduction

The P87LPC760 is a 14-pin single chip microcontroller designed for applications demanding high integration, low cost solutions over a wide range of performance requirements. It is based on an 80C51 processor architecture that executes instructions at twice the rate of standard 80C51 devices.

The P87LPC760 offers internal RC operation, wide operating voltage range, programmable I/O port configurations, LED drive outputs, two 16 bit timers, a build-in watchdog timer, four keypad interrupt inputs and power reduction modes.

All these features make the LPC760 very suitable for remote control transmitter applications. It is a very cost effective alternative for older or even discontinued devices like the PCA84C122 and the SAA3010.

3.2 Hardware

Figure 3.1 shows the main application of the P87LPC760 as a remote control transmitter. A 16-key pad, arranged as a 4 x 4 matrix, is implemented using only eight port pins of the microcontroller. The 'sense' (input) lines are connected to port 0. The P87LPC760 allows any pin of port 0 to be enabled to cause a single keyboard interrupt. The interrupt is generated when any enabled pin is pulled low by a key pressure.

The 'scan' (output) lines are designated to port 1 pins of the LPC760. By making each scan line logic 0 in turn, and each time looking at the sense lines, a depressed key can be detected. Each key of the transmitter keypad represents a corresponding command code, determined by using a software look-up table.

This code together with the system address is sent according the RC5 protocol. The pulses that are generated are available at port pin P1.7. This pin drives the output transistor, which provides the current for the IR-LED.

In our example, the on-chip reset and the micro's internal RC oscillator (6 MHz ± 5%) are used. Unused port pins could be used to expand the keypad matrix or for example to select the

controllers system address. If more I/O or on-chip program (code) memory is needed alternative microcontrollers, like the P87LPC761/2 are available from Philips Semiconductors.

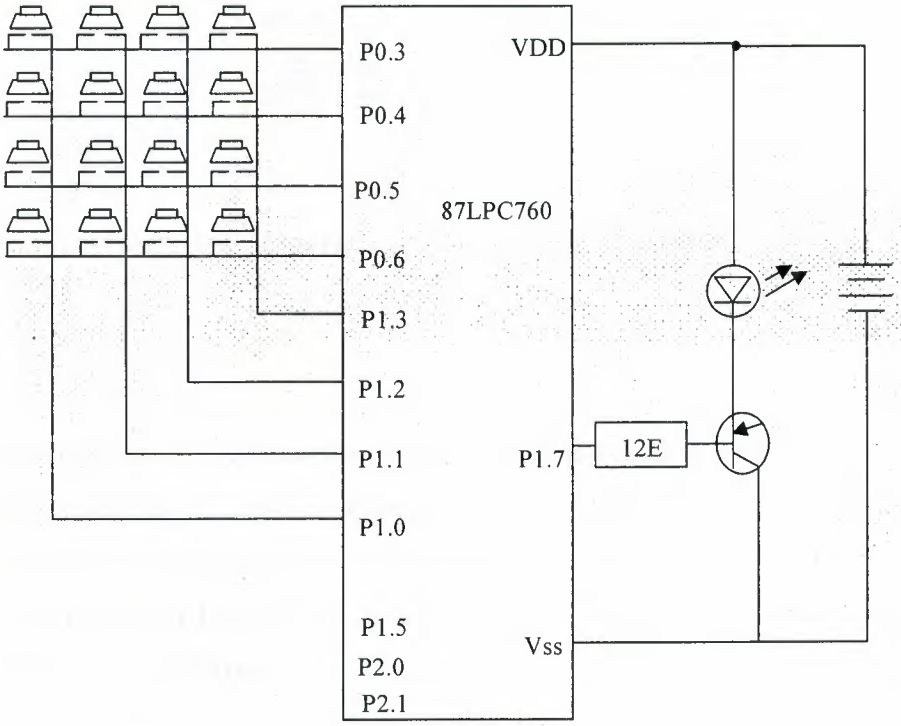


Figure 3.1 P87LPC760 Remote Control Transmitter Applications

3.3 RC5 Transmission Protocol

To ensure immunity to interference from other IR sources such as the sun, lamps and IR sound transmissions (to headphones), bi-phase encoding (also called Manchester encoding) is used for RC5 code words. As shown in figure 3.2 each bi-phase encoded bit is a symbol comprising two logic levels with a transition in the middle.

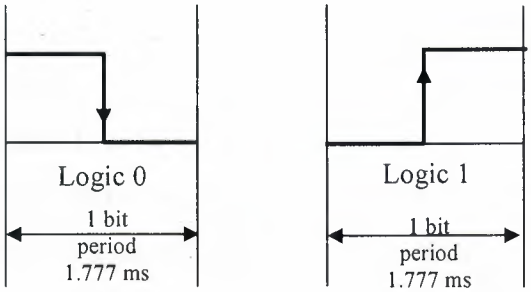


Figure 3.2 Bi-Phase Code Word Bits

As shown in figure 3.3, the bi-phase code words modulate a 36 kHz carrier, before being transmitted via the IR LED. Since the repetition period of the 36 kHz carrier is 27.778 us and the duty factor is 25 %, the carrier pulse duration is 6.944 us.

Because the high part of each bit of the RC5 code word contains 32 carrier pulses, 1 bit period is 64 x 27.778 us = 1.778 ms.

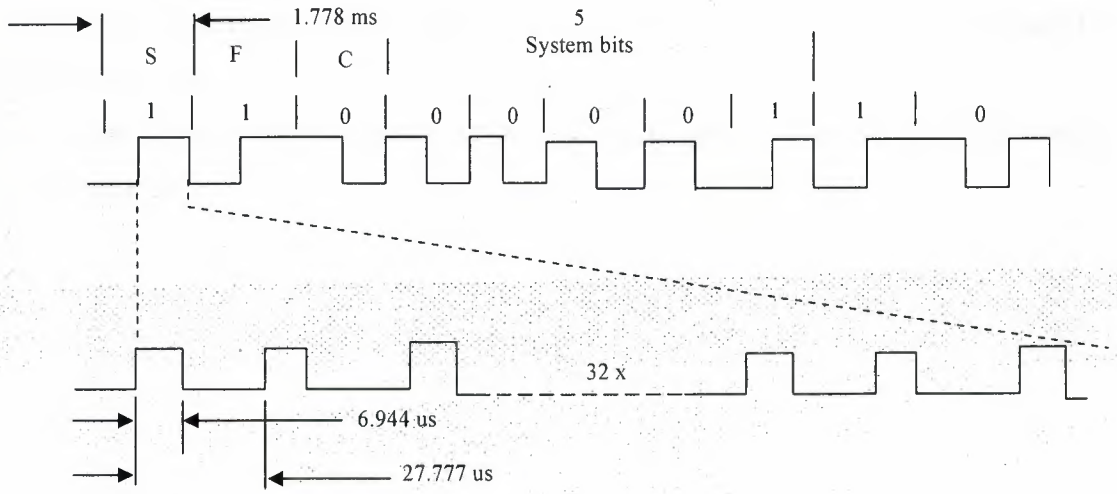


Figure 3.3 RC5 Code Word Examples

A complete RC5 code word contains 14 bits, so it takes 24.889 ms to transmit. Each 14 bit RC5 code word consists of:

- . a start bit (S) which is always logic 1
- . a field bit (F) which denotes command codes 0 to 63 or 64 to 127
- . a control bit (C) which toggles after each key release and initiates a new transmission
- . five system address bits for selecting one of 32 possible systems
- . six command bits representing one of the 128 possible RC5 commands

3.4 Software

3.4.1 Main loop

After initialization of the hardware, the four scan-lines (port pins P1.0-3) are pulled low and the LPC76x is forced into power down mode as shown in figure 3.4.

If one of the 16 keys is pressed a keyboard interrupt will be generated and the micro will wake up from power down mode. The main program checks for a valid key hit. Next, the control bit (C) of the system byte is set or reset. After that, a routine is called for sending out the key info as an RC5 code word.

Finally power down mode is entered again, waiting to wake up at the next keyboard or watchdog interrupt.

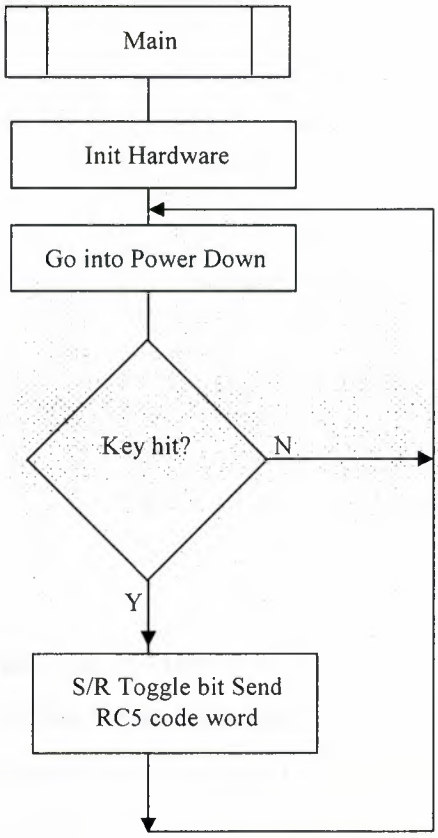


Figure 3.4 Main Software Loop

3.4.2 Keyboard Interrupt

When a key is pressed, an interrupt is generated. Inside the Keyboard Interrupt routine the keyboard interrupt is disabled and the de-bounce Timer 0 is started. This is done in order to let enough time (5 ms) for the key to stabilize so that we read a correct value from the port 0 sense lines. While waiting for the de-bounce timer to expire the micro is in idle mode to save power as shown in figure 3.5.

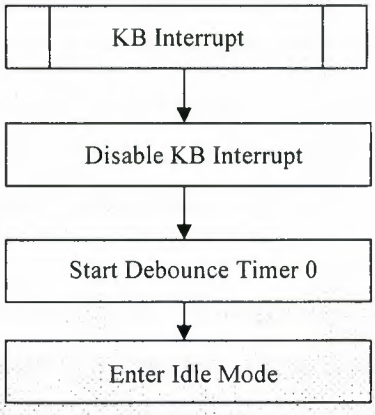


Figure 3.5 Keyboard Interrupt

3.4.3 Timer 0 (de-bounce timer) Interrupt

At the very beginning of the Timer 0 interrupt routine, after the timer is stopped a routine is called to decode the keypad. After detection of a valid key pressure this routine sets / resets the flag "key hit" (see figure 3.6).

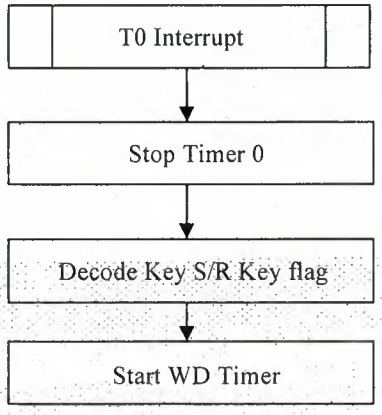


Figure 3.6 Timer 0 Interrupt

At the end of the timer interrupt the watchdog timer is started. This timer is used for repetition of sending RC5 code words (every 100 ms) as long as a key stays pressed.

3.4.4 Watchdog Timer Interrupt

If the watchdog timer overflows an interrupt is generated and the micro leaves power down mode (from inside the main loop). First the decode keyboard routine is called to check if a key is still pressed. If that is the case the watchdog is fed and the interrupt routine is left, resulting in sending a new RC5 command by the main loop and the next watchdog interrupt after approximately 100 ms.

If there is no more "key hit" (key released) the watchdog timer is stopped, the toggle bit is inverted and the keyboard interrupt is re-enabled. These results in the micro going back into power down mode, waiting for the next keyboard interrupt (key pressure).

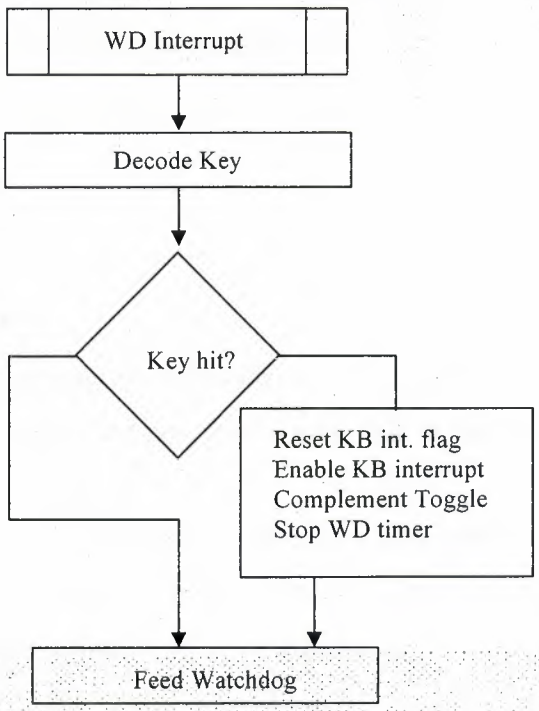


Figure 3.7 Watchdog Timers Interrupt

3.5 Send RC5 Code Word

The flow diagram below shows the routine used for sending out an RC5 code word. The program starts with setting a bit counter to 14 and sending the start bit. For sending out 36 KHz modulated bits the subroutine "Start_Mod" is called which is described later on in this document.

After transmission of the start bit, the next 13 bits of the RC5 code word are transmitted, using the above described Manchester encoding principle. After each bit transfer a delay of 889 microseconds is programmed using Timer 1 of the micro, according the RC5 specification. Than, the next two bits of the RC5 code word are checked to decide if a short (modulated) pulse, a long pulse or another extra delay should follow.

If the bit counter reaches zero the program returns back to the main loop and the micro enters power down mode again.

3.6 36 KHz Modulator

For the generation of 32 (or 64) pulses modulated at 36 KHz with a duty cycle of 25% Timer 1 of the micro is used. This part of the program is very time critical and is therefore written in assembly.

T1 is programmed in auto-reload mode generating an interrupt every 27.777 us (including latency). Inside the interrupt routine a pulse-counter is decremented and at port pin P1.7 a pulse of 7 us is generated, according the RC5 specification as shown in figure 3.8.

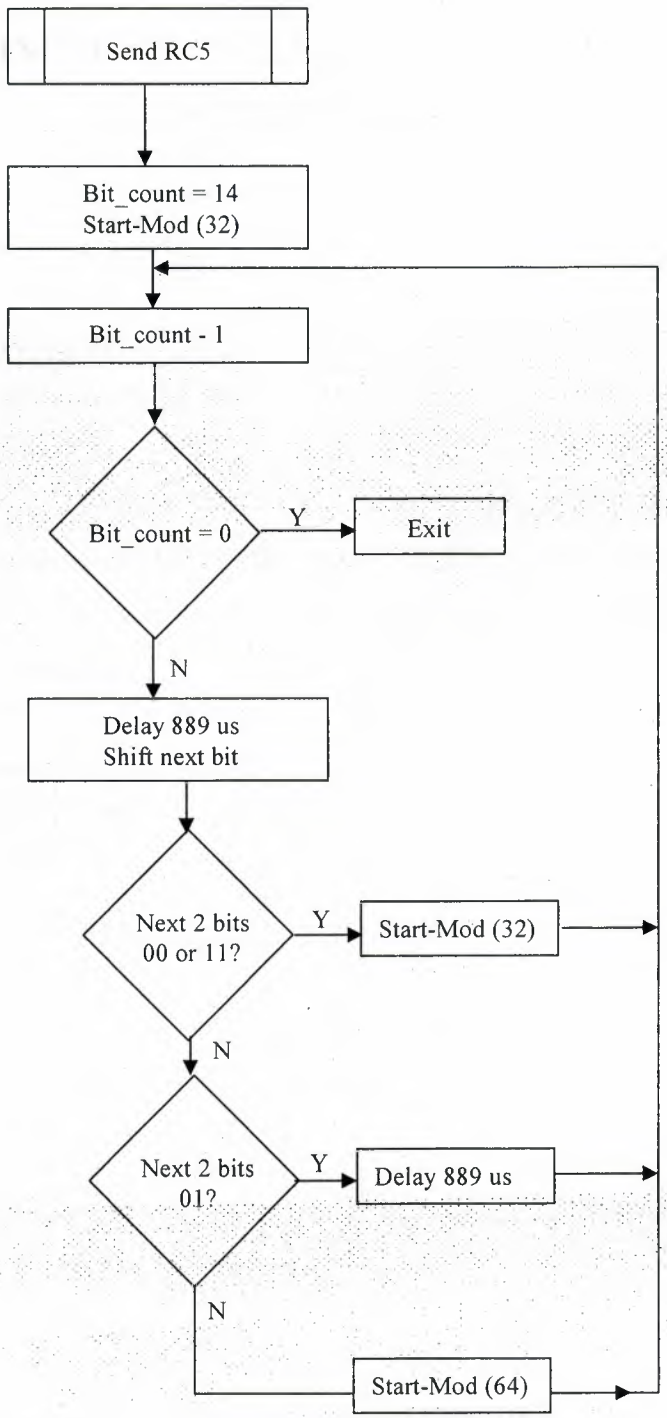


Figure 3.8 Generation of 32 (or 64) Pulses Modulated at 36 KHz

4. MEASURING DUTY CYCLES WITH AN INTEL MCS-51 MICROCONTROLLER

4.1 Introduction

The fastest way of measuring duty cycles is with the aid of hardware. The MCS-51 type of microcontrollers offers possibilities for that since they are equipped with two internal timer/ counters.

The two port-pins INT0 (P3.2) and INT1 (P3.3) can control these timer/ counters directly by hardware. Therefore we consider them as "fast-inputs". All other pins can control the timer/ counters only by software. The internal configuration of the hardware including the smart sensor SMT 160 is shown in figure 4.1.

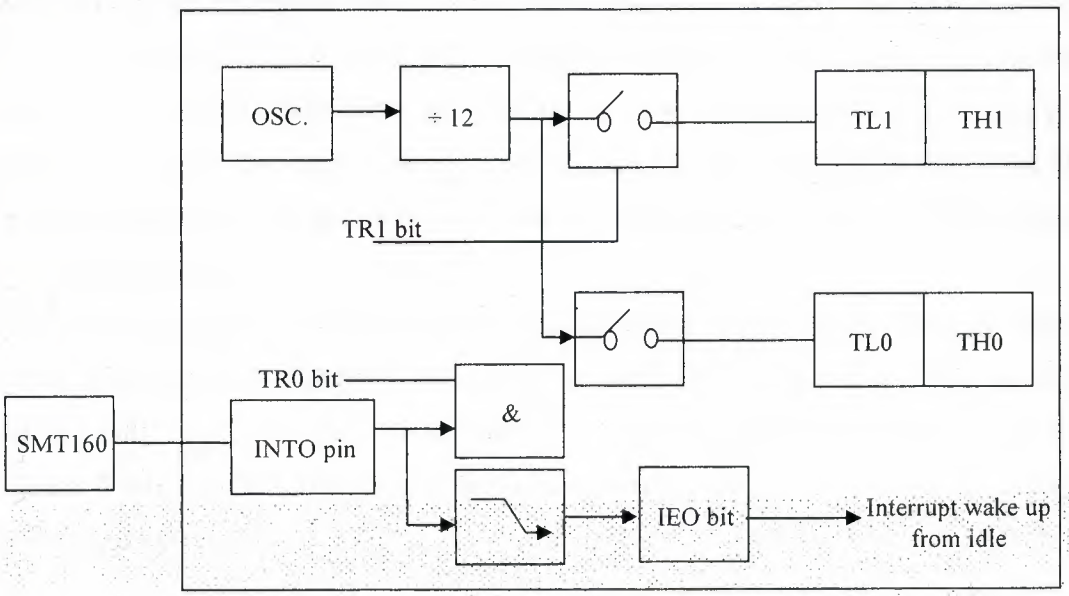


Figure 4.1 Configuration of Internal 8051 Hardware

This application note describes A) four assembly programs for the measurement of duty-cycles by hardware and B) a program for the measurement by software. Finally, we will discuss the conditions for which averaging can improve the resolution.

4.2 Hardware-controlled Measurement via INT0 Pin

When the duty-cycle is measured by hardware there are some restrictions concerning the tasks the CPU is performing: Both timers TIMER0 and TIMER1 are used. Normally TIMER1 is generating the baud rate for communication purposes. While measuring, the CPU is not allowed to transmit or receive any data.

Another restriction occurs when using this fastest and most accurate way of measuring, which is obtained by using interrupts preceded by the IDLE-mode of the CPU. This is important because, when the processing of an instruction would be interrupted, the instruction is first completed and not all instructions have an equal execution time. During the IDLE-mode, the CPU is non-active. The two timers are insensible to the IDLE command which is an important feature of the MCS-51. The CPU will start up again by an interrupt. On this way, the CPU responds maximally fast on an interrupt. This special way of measuring demands the CPU not to run any background programs because that will cause errors in both the measuring and the background program.

Both timer/ counters are selected to operate in the 16-bits timer mode. Therefore the "timer/ counter mode control" (TMOD) register is initialized with the value 19H. In this mode TIMER0 only runs when P3.2 is logically "1", while TIMER1 can only be controlled by software. TIMER1 measures the total measurement time. After a measurement the duty-cycle p is obtained by:

$$P = \frac{contents\_of\_TIMER0}{contents\_of\_TIMER1} \tag{4.1}$$

Detection of an edge with a resolution of 1µs is obtained when the measurement is started and stopped by using interrupts. An interrupt is generated on a falling edge of the input signal (when the interrupt flags are enabled).
The measurement is explained with the aid of a flow diagram (figure 4.2). Firstly, the contents of both timers are set. The initializing part starts with the detection of a 0-1 transition. Then the interrupt enable flag is set and the IDLE mode is invoked. Now the processor is waiting for an interrupt. When it is generated, which occurs at the next 1-0 transition of the input signal, the flags TR0 and TR1 in the "timer control" register (TCON) are set. Now TIMER0 only runs when P3.2 is pulled high, so it measures the time that the input signal is logically 1. TIMER1 is continuously running during the whole measurement. Note that TIMER1 starts 3µs too late because processing of an interrupt takes 3µs. However because the measurement will be stopped in the same way this delay is eliminated in the final result.

With respect to the measurement time, there is the choice to fix either the number of periods or the measurement time. Because the period duration can vary between 300µs and 800µs a fixed number of periods is an inefficient option. For short periods the measurement time is also short so the result will be troubled because of sampling noise. Therefore a fixed measurement time is chosen corresponding to 16 bits of machine cycles. Now the measurement will be finished after TIMER1 generates an overflow. In this case we have a 17 bits result which would require complex software routines. This problem is solved when TIMER1 is initialized (before the measurement) with an offset. This offset corresponds to the maximum length of two periods of the sensor signal (about 1.6ms in time). The offset is subtracted from the 17 bits result and will results in a 16 bits word.

When TIMER1 generates an overflow the measurement has to be stopped (see the right-hand branch in figure 4.2. After occurrence of the next 1-0 transition of the sensor signal the interrupt-enable flag is set and once again the IDLE mode is invoked. After occurrence of the interrupt the flags TR0 and TR1 in the TCON register are cleared.
After correction for the offset in TIMER1, the contents of both timers are used to calculate the duty-cycle.

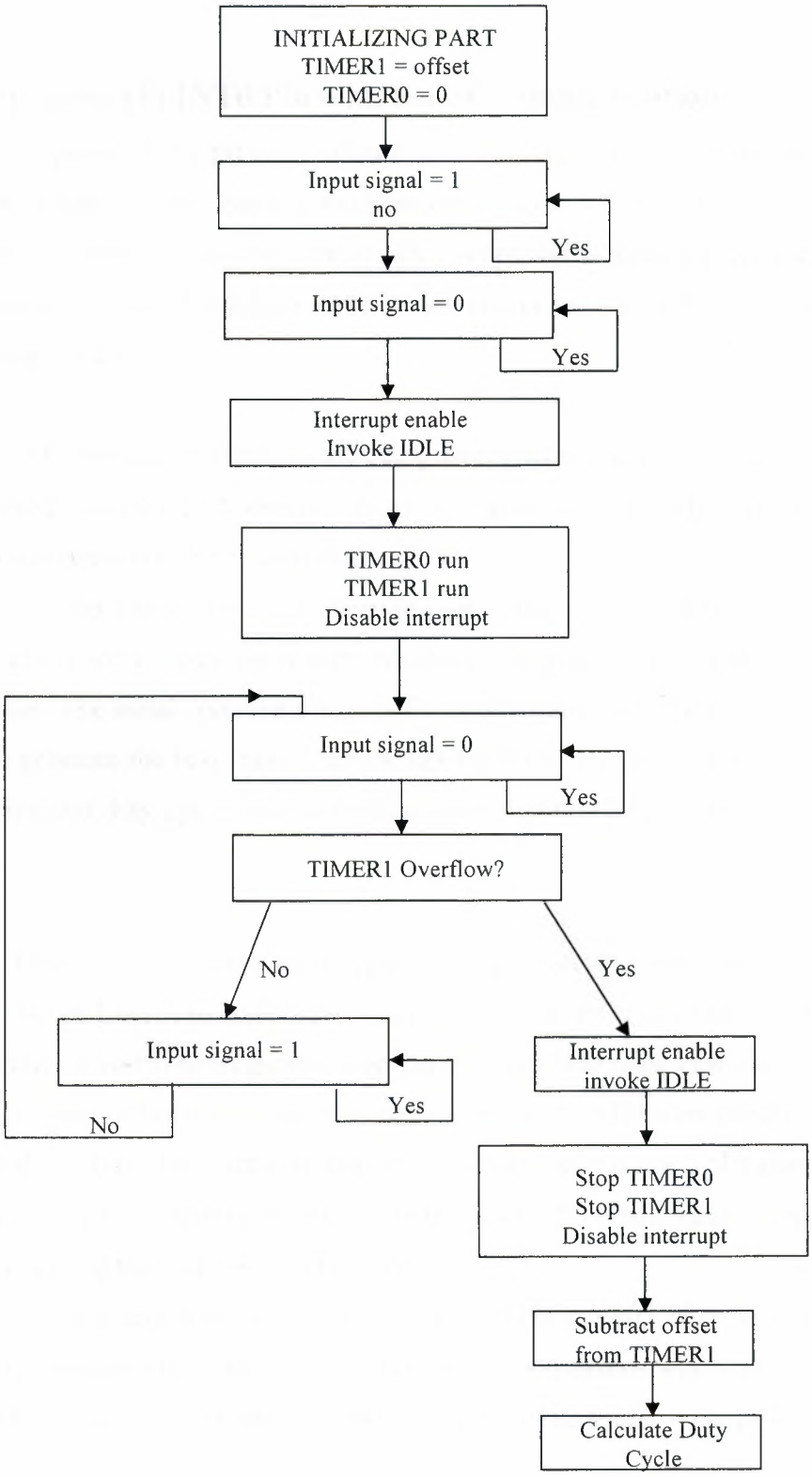


Figure 4.2 Flow Diagram of a Duty-Cycle Measurement with a Resolution of One Machine Cycle

4.3 Measurement via INT0 Pin with Serial Communication

The method proposed in the previous section uses the 8051 IDLE mode to create a constant delay (interrupt latency) between the moment of interrupt (on the falling edge of the input signal) and the moment of sampling Timer0. This is necessary because when the processing of an instruction is interrupted, the instruction is first completed and not all instructions have an equal execution time.

During the IDLE mode no instructions are being processed because the execution unit of the 8051 is disabled. However, the interrupt timer and serial units are left running. In this mode the power consumption is significantly reduced.

To realize a constant latency only one interrupt source may be enabled (as is the case in figure 4.1). However, in some cases we require simultaneous temperature measurement and serial communication. The serial communication is likely to require and interrupt of its own as well as a timer to generate the baud rate. Using a 33MHz 8051 it is possible to realize 1200 baud communication and duty-cycle measurement, without additional hardware, as shown in figure 4.3.

In this case Timer 1's overflow rate is required to generate the baud rate. Using a 32.9856 MHz crystal Timer 1 needs to count 859 clocks to overflow. Proposed that it is used in a 16 bit mode. The Timer 1 overflow bit generates an interrupt (TF1) to reload the divider value.

Since we are using a fast processor this same interrupt handler can increment a software counter in a short time. This software counter combined with the actual value of Timer 1 is used as a time base to determine the period of the SMT160 output signal. Timer 0 retains its function for counting the high period of the signal.

Of course, since we are now using 3 interrupt handlers (INT0, TF1, Serial) the interrupt latency is not constant anymore, so the resolution of the measurement will be degraded by a factor of 3. However, this is compensated by the higher clock speed of the processor.

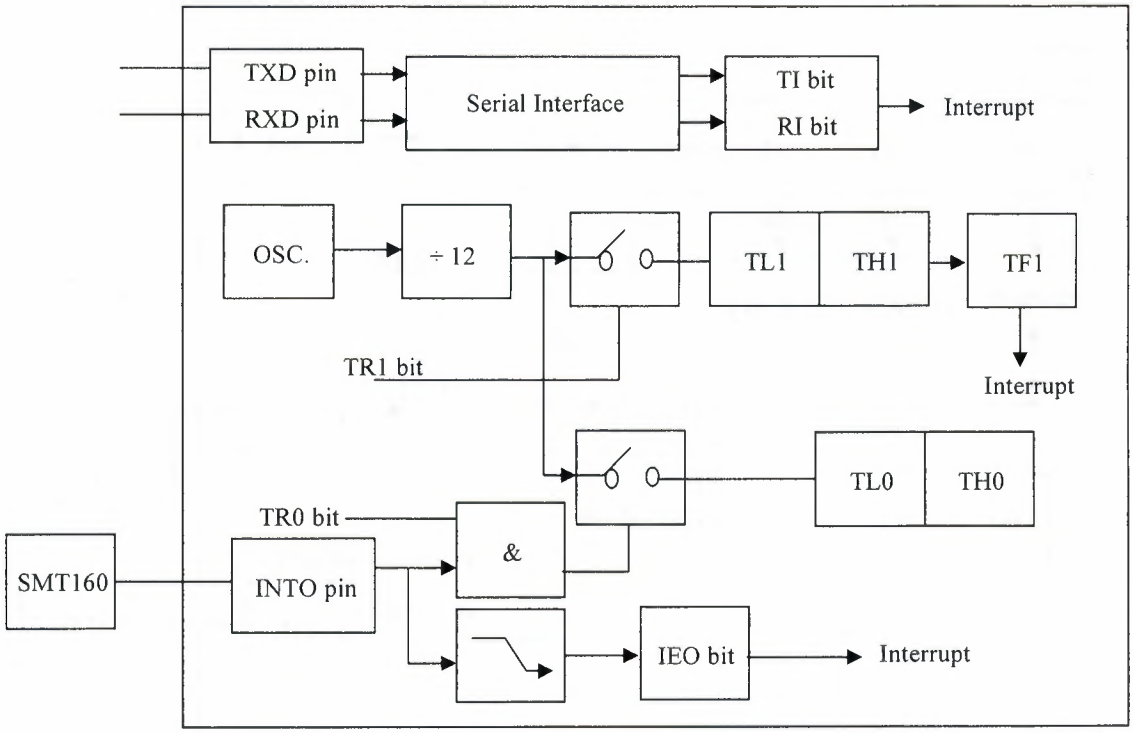


Figure 4.3 Simultaneous Measurement and Serial Communication

4.4 Duty-Cycle Measurement using Timer 2; Capture Register

Many 8051 derivatives, including 8052 and the 16 bit 8051XA, are equipped with an additional timer, Timer 2 figure 4.4. Timer 2 is an advanced 16 bit timer/counter with capture/reload register. In our case, the function of the capture register is to instantaneously load the value of Timer 2 (capture) and hold it until the interrupt handler reads it. This eliminates the effect of the interrupt latency, provided that the latency is less then the interrupt rate.

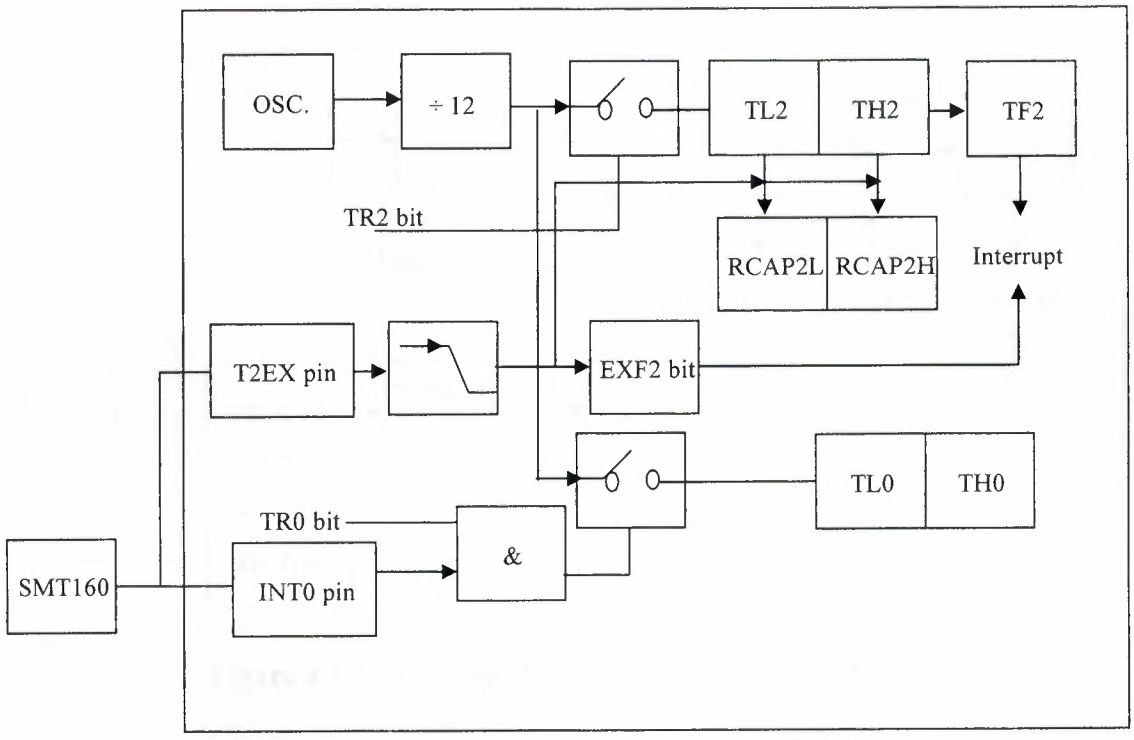


Figure 4.4 Duty-Cycle Measurement with Timer 2 and Timer 0

Using Timer 2 for the measurement of the period of the SMT160 signal and Timer 0 for the high period, Timer 1 is free to be used as a baud rate generator for the serial interface.

Sometimes, Timer 0 can not be spared for the measurement of the high period of the SMT160 signal, for instance when real time operating system (RTOS) is used. The scheduler of the RTOS often requires a clock to generate the time slices of each process.

In that case Timer 0 might be in use by the RTOS. By adding an external XOR gate figure 4.5, Timer 2 will be sufficient to measure the duty-cycle of the SMT160. By toggling pin OUT in the interrupt handler of Timer 2, both rising and falling edges can be captured.

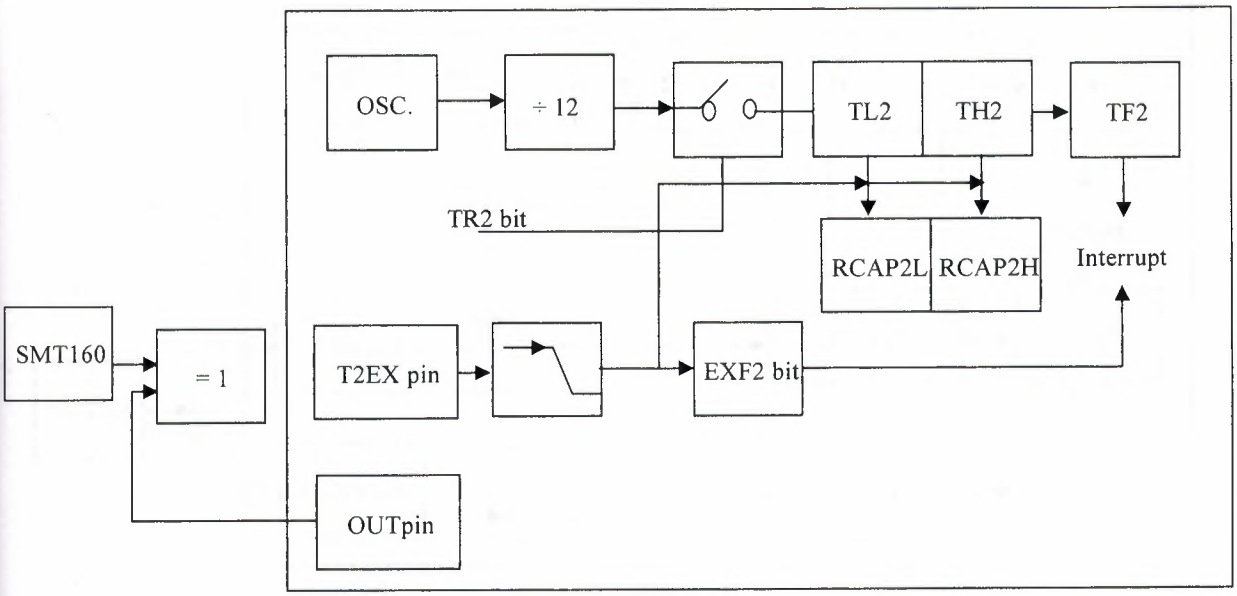


Figure 4.5 Duty-Cycle Measurement using Timer 2 Only

4.5 Duty-Cycle Measurement using a Programmable Counter Array (PCA)

The 8051FX derivatives are equipped with an additional piece of hardware: the programmable counter array. This consists of one timer and 5 capture registers.

The timer can be programmed to run at a frequency Osc/12 or Osc/4. As compared to the ordinary 8051, the use of the FX types enables to perform the measurements with 3 times the resolution in the same measurement time.

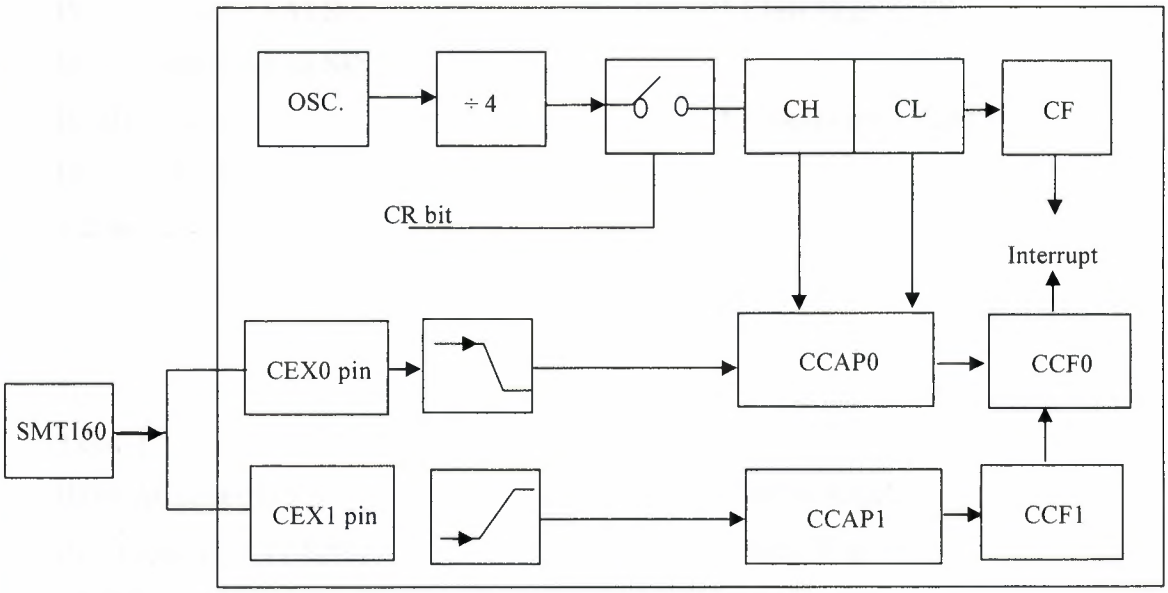


Figure 4.6 Duty-Cycle Measurement using the PCA

Moreover, the capture registers can be programmed to capture on rising or falling edges, or both, so no external XOR gates are required. Since there is a capture register available for both the rising and the falling edge, interrupt latencies are non critical using this processor family figure 4.6. This means the interrupts handlers can be easily written using a high level programming language like C.

An example of an interrupt handler that measures *n* periods of the SMT160 signal consecutively is given below.

```
void PCAHandler(void) interrupt 6 using 1 {
static union Word2Byte CaptureUp, CaptureDo;
if (PCAOverFlow) {                          /* PCA Overflow? */
PCAOverFlow = FALSE;
if (!Ready) {
if (OverFlow > 3) {                         /* 3 overflows => error */
SetCaptureOff();                            /* Capture off */
```

```
PCACapture0 = FALSE;                    /* Clear flags */
PCACapture1 = FALSE;
Ready = TRUE;                           /* measurement done */
Error = TRUE;
} else {
OverFlow++;
};
};
} else {
if (PCACapture1) {                      /* rising edge? */
PCACapture1 = FALSE;                    /* Clear flag */
if (!Ready) {
CaptureDo.Byte.Hi = CCAP1H;             /* save PCA value */
CaptureDo.Byte.Lo = CCAP1L;
HiTime += (CaptureDo.Word - CaptureUp.Word);
};                                      /* determine low period */
} else {
PCACapture0 = FALSE;
if (!Ready) {
CaptureUp.Byte.Hi = CCAP0H;             /* save PCA counter */
CaptureUp.Byte.Lo = CCAP0L;
if (First) {
First = FALSE;                          /* 1st time just caputure value */
HiTime = LoTime = 0;
SetPCA1NegEdge();                       /* enable falling edges */
} else {
LoTime += (CaptureUp.Word - CaptureDo.Word);
if (--Count == 0) {                     /* when Count = 0 ready */
SetCaptureOff();                        /* capture off */
PCACapture0 = FALSE;                    /* clear flags */
```

```
PCACapture1 = FALSE;
PCAOverFlow = FALSE;
Ready = TRUE;                                   /* measurement ready */
                    };
               };                               /* determine high period */
          };
};
if (!Ready) {
};
OverFlow = 0;                                   /* we have a signal */
};
return;
}
```

We interface with the interrupt handler from the main program, using the following functions:

```
#include <pca.h>
#include <stdio.h>
#define PERIODS 25
struct DoubleByte {
unsigned char Hi, Lo;
};
union Word2Byte {
unsigned short Word;
struct DoubleByte Byte;
};
static volatile bit First, Ready, Error;
static volatile unsigned int Count = 0, Periods = 51;
static volatile unsigned char OverFlow;
static volatile unsigned long HiTime, LoTime;
void StartCount(void) {
First = TRUE;                                   /* Initializes all varialbes */
```

```
OverFlow = 0;
Count = Periods;
Ready = FALSE;
Error = FALSE;
SetPCA0PosEdge();                    /* Enable capture */
}
void SetPeriods(unsigned APeriods) {
Periods = APeriods;
}
bit IsReady(void) {
return(Ready);
}
bit IsError(void) {
return(Error);
}
float GetDutyCycle(void) {
return (float)HiTime / (HiTime + LoTime);
}
```

The main program needs to initialize the interrupt handler once, the repeatedly start a measurement, wait till the measurement is finished, check for errors and display the result.

```
main() {
SetPeriods(51);
while(TRUE) {
StartCount();
while(!IsReady()); continue;
if(IsError()) printf("An error has occured\n");
else printf("The termperature is %f\n",
(GetDutycycle() - 0.32) / 0.0047));
        }
}
```

4.6 Software-Controlled Measurement

Only the I/ O ports P3.2 and P3.3 can be used to detect interrupts. Therefore, when sensors are connected to the other I/ O ports only a software-controlled measurement can be used to measure the duty-cycle. Again two counters are required. However, it is still possible to use a hardware timer, although it is software controlled. This timer TIMER0 can count the measurement time. A fast software routine is used to measure the "1" state of the sensor signal. The results are stored in a counter called: HIGH_COUNTER.

The timer TIMER0 increments every machine cycle, which takes 1μs. The software sample rate takes 3μs. Therefore, to obtain the duty-cycle *p*, HIGH_COUNTER is multiplied by 3, according to the equation:

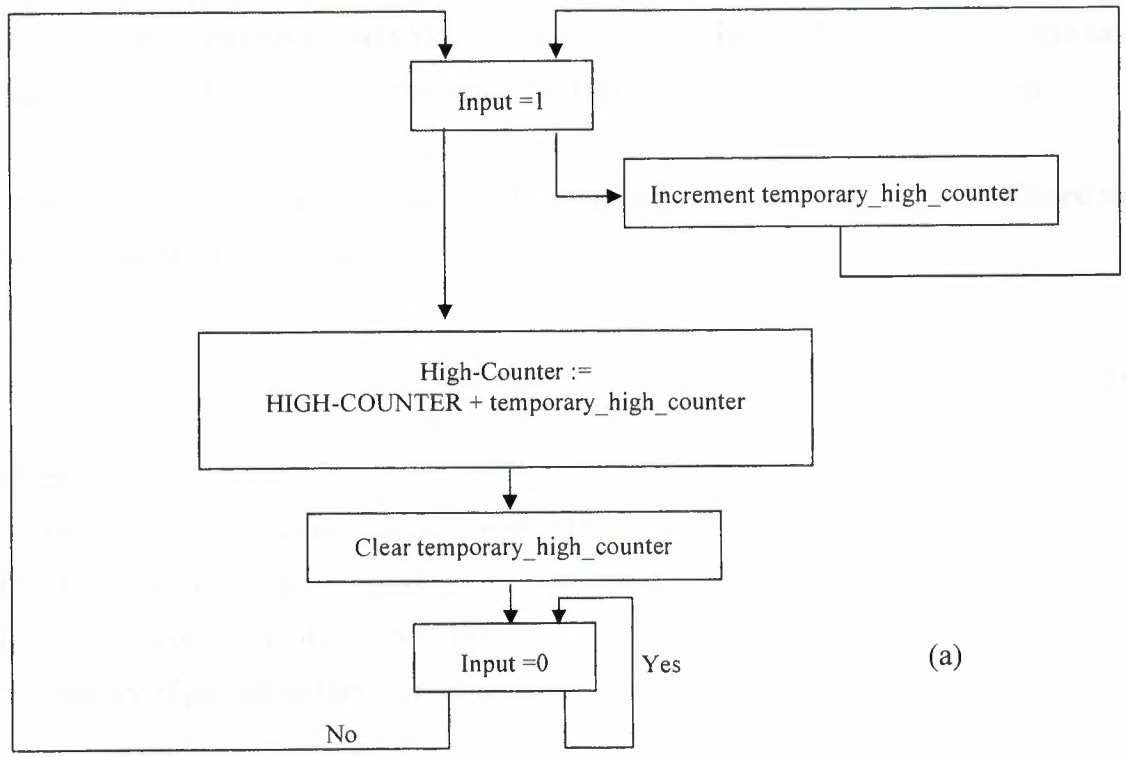
$$P = \frac{3 \times HIGH\_COUNTER}{TIMER0} \tag{4.2}$$

Normally HIGH_COUNTER should store more than 8 bits and therefore requires two 8-bits registers. This would cause a decrease of the sampling rate, because two extra commands would be needed to "glue" these registers (test on overflow of the low byte and, depending on the test result, incrementing of the high byte). Therefore, an alternative solution has been applied: When the input signal is low, HIGH_COUNTER is waiting until the signal goes high again. This time can be used to "calculate" HIGH_COUNTER figure 4.7.

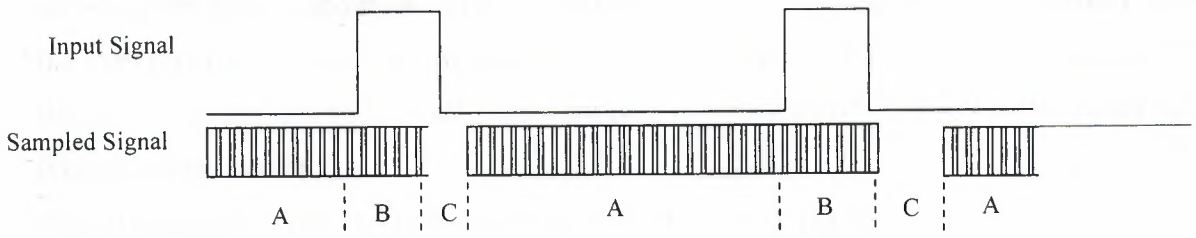
This figure shows the use of a temporary-result register which is called: temporary high counter. This counter contains the number of samples for which the input signal was high during one period. As soon as the input signal goes low, the value of HIGH_COUNTER is calculated by adding the temporary high counter to it. During this calculation interval the sensor signal is not sampled.

This restricts the duty-cycle to a limit. However the calculations take only 15μs, so that even when the duty-cycle equals 0.95 for a period of about 600 μs, there will not be a problem.

The counts for the measurement time are stored in the hardware timer/ counter TIMER0. It is started and stopped by software at a 1-0 transition of the input signal. In that case



(a)



(b)

Figure 4.7 a) Flow Diagram of a Duty-Cycle Measurement by Software (only the Part to Measure the "High"-Time). b) Time Diagram Belonging to figure 4.7(a). During Interval C HIGH_COUNTER is Calculated Followed by Clearing of the Temporary_High_Counter

Temporary high counter doesn't have to count so this action is of no influence on the sample rate. The speed of incrementing the temporary high counter (the sampling rate) is 3µs.

Examples: The standard deviation σ of the sampling noise, of a duty-cycle modulated signal can be calculated from the equation:

$$\sigma = \frac{t_s}{\sqrt{6T_m \times T_p}} = \frac{1}{\sqrt{6N}} \times \frac{t_s}{T_p} \qquad (4.3)$$

where:

ts= the time interval between successive samples

TP = the period of the input signal

Tm = the measurement time (= N × Tp)

N = number of periods within 1 measurement

The period of the input signal is between 300µs (at 40^0C) and 800µs (at -40^0C or 120^0C). The measurement time is about 64ms (slightly smaller than $2^{16} \times 1$µs because of the offset). When the sampling rate is 1 µs, then the sampling noise is between ó ≡ 5×10^{-5} and 10^{-4}. As a rule of thumb we can say that 95% of all values are read in the range of ± 2σ around the mean value (Gaussian distribution).

When the sampling rate is 3µs the sampling noise is 3 times more:

4.7 Understanding Averaging of Measurement Results

Using an ordinary A/D converter averaging successive measurements will not yield an improved resolution. With duty-cycle measurements the resolution can be improved by averaging successive measurement results - under certain conditions.

Obviously the successive measurement results should not be correlated. This is true when the period of the period SMT160 is not an integer multiple of the period of the microcontroller's timer. We can ensure this by measuring the jitter of the falling edges using a frequency counter, with the gate time set to τ ms. This jitter appears to be a function of τ. This effect can also be visualized on an oscilloscope with DTB function. This function will allow you to zoom

in on the nth falling edge after the trigger. As you can see, the jitter increases with n. When this jitter is larger then one period of the microcontroller's timer, successive duty-cycle measurements will not be correlated. This means the resolution will increase with the square root of the number of samples (as described elsewhere), when a certain minimum delay between successive measurements is observed.

Figure 4.8 was actually measured using a microcontroller with 1.25 MHz clock. Using this setup, the quantization noise approximately equals the thermal (and other) noise with a 1 ms interval between measurements. This means that for uncorrelated duty-cycle measurements a 1 ms interval must be observed. In another setup the noise might show different behavior due to electromagnetic interference etc.

From the figure it can also be estimated that with a 4 MHz clock (for instance a 8051FA running at 16 MHz and using the PCA) a zero delay between measurements can be used, thus obtaining maximum measurement speed.

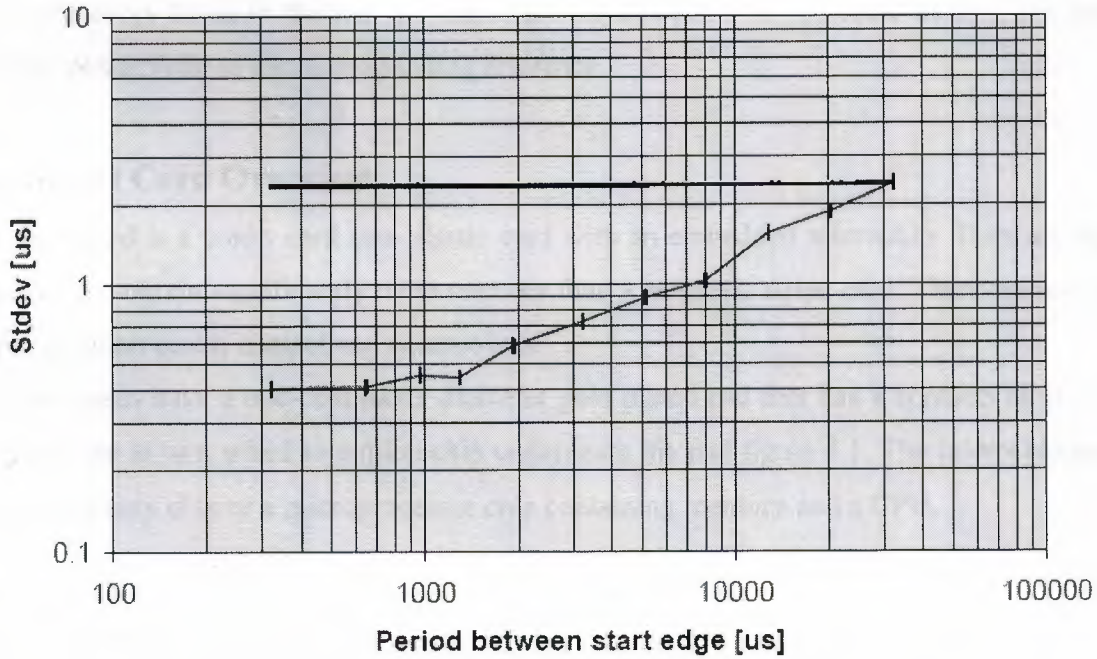


Figure 4.8 Jitter as a Function of the Gate Time

5. SMART CARDS AND THEIR OPERATING SYSTEMS

5.1 Introduction

The smart card is one of the latest additions to the world of information technology. The smart card has a microprocessor or memory chip embedded in it. The chip stores electronic data and programs that are protected by advanced security features. When coupled with a reader, the smart card has the processing power to serve many different applications. Smart cards provide data portability, security and convenience.

Smart cards currently are used in telephone, transportation, banking, and healthcare transactions, and soon to be used in Internet applications. Smart cards are already being used extensively in Japan and Europe and are gaining popularity in the U.S. In fact, the development of the smart card industry is very fast.

The references listed in the end are important for this paper to the same extend. For further details, please refer to the corresponding reference.

5.2 Smart Card Overview

A smart card is a credit card size plastic card with an embedded microchip. They are highly secure and contain significantly more memory than a magnetic stripe card. There are two basic types of smart cards: contact and contact-less.

Contact cards have a one-centimeter diameter gold plated pad that has 8 contacts on it. These contacts are in turn wired to a microchip underneath the pad figure 5.1. The microchip can be a memory only chip or a microprocessor chip containing memory and a CPU.

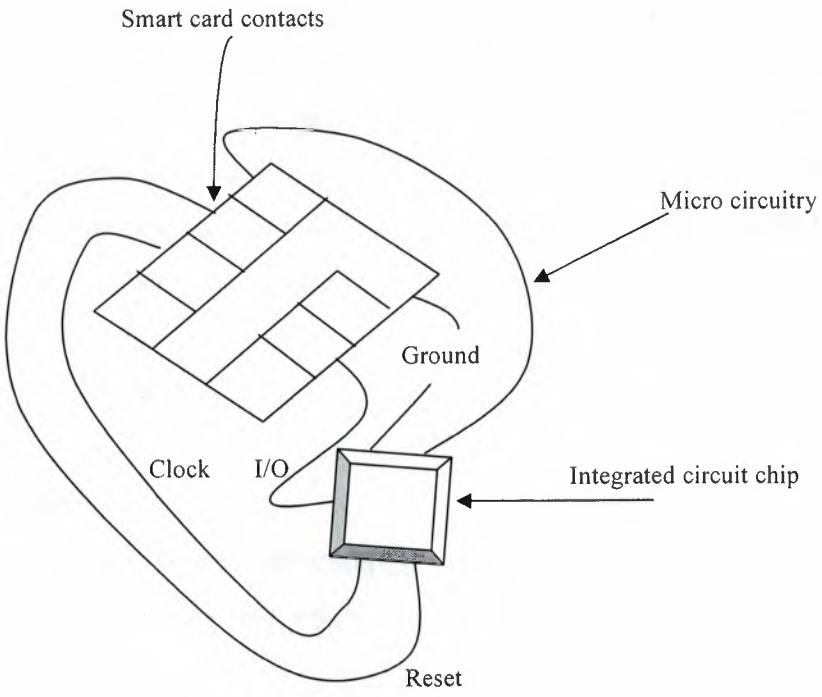


Figure 5.1 Smart Card Contacts

Memory cards are used mostly as telephone cards whereas microprocessor cards can be used for multiple applications on the same card. Although both cards can have stored value and stored data areas, the microprocessor card can in addition process the data since it contains a CPU, RAM and an operating system in read only memory (ROM).

Contact-less cards have an embedded microprocessor chip but also contain a miniature radio transceiver and antenna. They only operate within close proximity to the reader.

Instead of inserting the card you simply pass the card close to the reader. Contact-less cards tend to be more costly than contact cards and are best suited for transportation and building access applications Magnetic stripe cards have no processing capability and are limited to less than 100 bytes of memory. They are significantly less secure than smart cards but they do cost less. However, magnetic stripe card readers are often 10 times more expensive than smart card readers and are less reliable.

Hybrid cards can be any combination of contact, contact-less and magnetic stripe cards.
Since 1982 the French banks have used the combination of chip and magnetic stripe cards as a bank credit card, allowing the banks to migrate to smart chip cards (Scott Guthery & Tim Jurgensen, 1998). They get the advantage of a more secure card while allowing a reasonable time to upgrade locations from magnetic stripe. There are even some hybrid cards that contain a microchip, magnetic stripe, bar code, optical code, picture and signature panel all in one card.

5.3 Smart Card Hardware

The computer on a smart card is a single integrated circuit chip that includes the central processing unit (CPU), the memory system, and the input/output lines figure 5.2.

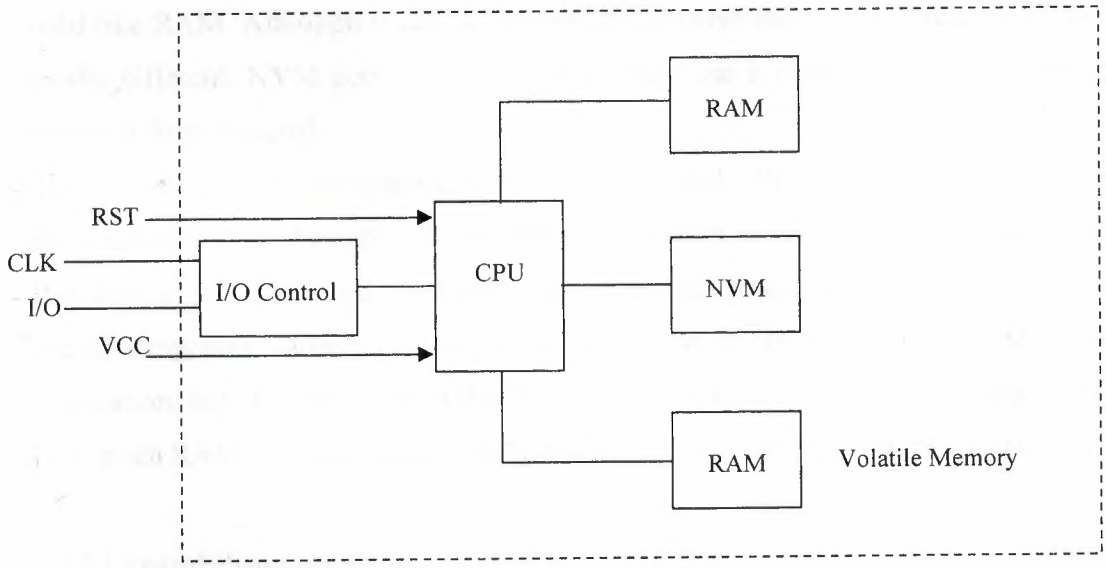


Figure 5.2 Elements of a Smart Card Computer System

5.3.1 Memory System

Smart cards have a memory architecture that will be unfamiliar to most mainstream programmers. In fact, there are three kinds of memory on a smart card: read-only memory (ROM), nonvolatile memory (NVM), and a relatively tiny amount of random access memory (RAM). See figure 5.2.

Read-only memory is where the smart card operating system is stored. General-purpose, here, one finds various utility routines such as doing communication and for maintaining an on-card file system along with encryption routines and special-purpose arithmetic routines. Code and data are placed in read-only memory when the card is manufactured and cannot be changed; this information is hardwired into the card.

NVM is where the variable data such as account numbers, number of loyalty points, or amount of e-cash is stored. NVM can be read and written by application programs, but it cannot be used like RAM. Although it can be written, the purpose and the performance of the action is totally different. NVM gets its name from the fact that it retains its contents when power is removed from the card.

There is some RAM on a smart card, but not very much. This is the most precious resource on the smart card from the card software developer's point of view. Even when using a high-level language on the smart card, the programmer is acutely aware of the need to economize on the use of temporary variables. Furthermore, the RAM is not only used by the programmer's application, but also by all the utility routines, so a programmer has to be aware not only of how much RAM he is using, but also how much is needed by the routines he calls.

5.3.2 Central Processing Unit

For earlier 8-bit microcontroller, the central processing unit in a smart card chip is typically using the Motorola 6805 or Intel 8051 instruction set. These instruction sets have the usual complement of memory and register manipulations, addressing modes, and input/output operations. CPUs execute machine instructions at the rate of about 400,000 instructions per second (400 KIP), although speeds of up to 1 million instructions per second (1 MIP) are becoming available on the latest chips. The demand for stronger encryption in smart cards has

outstripped the ability of software for these modest computers to generate results in a reasonable amount of time. Typically 1 to 3 seconds is all that a transaction involving a smart card should take; however, a 1024-bit key RSA encryption can take 10-20 seconds on a typical smart card processor. As a result, some smart card chips include coprocessors to accelerate specifically the computations done in strong encryption. (Scott Guthery & Tim Jurgensen, 1998) After more than 20 years development, smart cards are evolving quickly. For example, memory sizes are increasing and processor architectures are moving from 8-bit to 16-bit and 32-bit configurations.

5.3.3 Smart Card Input/Output

The input/output channel on a smart card is a unidirectional serial channel. The smart card hardware can handle data at up to 115,200 bps, but smart card readers typically communicate with the card at speeds far below this.

The communication protocol between the host and the smart card is based on a master (host) and slave (smart card) relationship. The host sends commands to the card and listens for a reply. The smart card never sends data to the host except in response to a command from the host.

5.4 Smart Card Software

There are fundamentally two types of smart card software. One is host software, which is software that runs on a computer connected to a smart card. Host software is also referred to as reader-side software. The other is card software, which is software that runs on the smart card itself. As a counterpart of reader-side software, card software is also referred to as card-side software.

Most smart card software is host software. It is written for personal computers and workstation servers, accesses existing smart cards and incorporates these cards into larger systems. Host software will typically include end-user application software, system-level software that supports the attachment of smart card readers to the host platform, and system-level software that supports the use of the specific smart cards needed to support the end-user application. In

addition, host software includes application and utility software necessary to support the administration of the smart card infrastructure.

Host software is usually written in one of the high-level programming languages found on personal computers and workstations C, C++, Java, BASIC, COBOL, Pascal, or FORTRAN and linked with commercially available libraries and device drivers to access smart card readers and smart cards inserted into them. In contrast, card software is usually written in a safe computing language such as Java, machine-level language such as Forth, or assembly language.

Host application software sometimes substitutes the smart card for an alternative implementation of the same functionality (for example, when an encryption key or a medical record is kept on a smart card rather than on a hard disk file on the local computer or in a central database on a server). Host system software manipulates the unique computing and data storage capabilities of the smart card by sending commands and data to it and by retrieving data and results from it Card software is usually classified as operating system, utility, and application software, much as is the case with host software.

Card application software is typically used to customize an existing smart card for a particular application and amounts to moving some functionality from host application software onto the card itself. This may be done in the interest of efficiency in order to speed up the interaction between the host and the card or security in order to protect a proprietary part of the system. Card system software is written in a low-level machine language for a particular smart card chip and is used to extend or replace basic functions on the smart card.

Host application and card application are fundamentally different in their orientation and outlook. Card software focuses on the contents of a particular card. It provides computational services for applications in accessing these contents, and protects these contents from many applications, which might try to access them incorrectly. Host software, on the other hand, might make use of many different cards. It is typically aware of many cardholders and possibly many card issuers as well as many different kinds of cards.

Card software implements the data and process security properties and policies of a particular smart card. For example, a program running on the card might not provide an account number stored on the card unless presented with a correct personal identification number (PIN). Or a

program running on the card might compute a digital signature using a private key stored on the smart card, but it would under no condition release the private key itself. Software running on a smart card provides secure, authorized access to the data stored on the smart card. It is only aware of the contents of a particular smart card and entities such as people, computers, terminals, game consoles, set-top boxes, etc. trying to get at these contents. Host software connects the smart cards and the users carrying them to larger systems. For example, software running in an automatic teller machine (ATM) uses the smart cards inserted by the bank's customers to identify the customer and to connect the customers with their bank accounts. Host software is aware of many smart cards and tailors its response based on the particular smart card presented. Unlike most computer software, which relies on supporting services from its surrounding context, smart card software begins with the assumption that the context in which it finds itself is hostile and is not to be trusted. Until presented with convincing evidence to the contrary, smart cards don't trust the hosts they are inserted into and smart card hosts don't trust cards that are inserted into them. A smart card program only trusts itself. Everything outside the program has to prove itself trustworthy before the program will interact with it.

So it is useful to occasionally further categorize smart card software into application software or system software. (Scott Guthery & Tim Jurgensen, 1998) Application software uses the computational and data storage capabilities of a smart card as if they were those of any other computer and is relatively unaware of the data integrity and data security properties of the smart card. System software, on the other hand, explicitly uses and may contribute to and enhance the data integrity and data security properties of the smart card.

5.5 Smart Card Standard

The basic contact smart card standard is the ISO 7816 series, part 1-10 while contactless cards will be governed by the ISO 14443 standard. These standards are derived from the identification card standards and detail the physical, electrical, mechanical, and application programming interface. Below is a list of the contact card standards (Smart Card Industry Association, 2000).

1. IS 7816 - 1 (1987): Physical characteristics
   - Amendment 1 (1998) : Revised edition March 1998
2. IS 7816 - 2 (1988): Dimension and location of contacts
   - Revised edition March 1998
3. IS 7816 - 3 (1989): Electronic signals and transmission protocol
   - Amendment 1 (1992) : Protocol T=1
   - Amendment 2 (1994) : Revision of Protocol Type Selection
   - Amendment 3 (1998) : Introduction of 3 Volts ICCs
4. IS 7816 - 4 (1995): Inter-industry commands and responses
   - Amendment 1: (1998) : Revision Secure Messaging
5. IS 7816 - 5 (1994): Registration system for application identifiers
   - Amendment 1 (1996) : Registration of identifiers
6. IS 7816 - 6 (1995): Data elements for interchange
   - Amendment 1 (DIS) : Registration of IC Manufacturers
7. IS 7816 - 7 (1998): Smart Card Query Language commands
8. DIS 7816 - 8: Inter-industry Security Commands
9. CD 7816 - 9: Inter-industry Enhanced Commands
10. ISO 7816 -10 (1999): Synchronous cards

5.6 Smart Card Operating System

The smart card's Chip Operating System (frequently referred to simply as COS; and sometimes referred to as the Mask) is a sequence of instructions, permanently embedded in the ROM of the smart card. Like the familiar PC DOS or Windows Operating System, COS instructions are not dependent on any particular application, but are frequently used by most applications.

Chip Operating Systems are divided into two families:

- The general purpose COS which features a generic command set in which the various sequences cover most applications, and
- The dedicated COS with commands designed for specific applications and which can even contain the application itself. An example of a dedicated COS would be a card designed to specifically support an electronic purse application.

The baseline functions of the COS which are common across all smart card products include:

- Management of interchanges between the card and the outside world, primarily in terms of the interchange protocol.
- Management of the files and data held in memory.
- Access control to information and functions (for example, select file, read, write, and update data).
- Management of card security and the cryptographic algorithm procedures.
- Maintaining reliability, particularly in terms of data consistency, sequence interrupts, and recovering from an error.
- Management of various phases of the card's life cycle (that is, microchip fabrication, personalization, active life, and end of life).

The basic relationship between a smart card terminal and the smart card itself is of master and slave. The terminal sends a command to the smart card, the smart card executes the command, returns the result if any to the terminal, and waits for another command.

In addition to describing the physical characteristics of a smart card and the detailed formats and syntaxes of these commands and the results they return, smart card standards such as ISO 7816 and CEN 726 also describe a wide range of commands that smart cards can implement. Most smart card manufacturers offer smart cards with operating systems that implement some or all of these standard commands, together with manufacturer-specific extensions and additions. Earlier in COS evolution, the issuer has to commit to a specific application developer, operating system and chip for each service the issuer wished to provide to its customer base. This leaves almost no flexibility to change any of these components without

having to invest funds into a new software and/or hardware implementation. As a result early smart cards were costly and inflexible.

But today we can clearly see a development towards open operating systems that support multiple applications. For on-card application development of programs that run inside the secure environment of the smart card chip, we highly recommend operating systems that have bigger market exposure such as JavaCard OS, MultOS and lately Windows for smart cards (JCI Smart Card System Consulting, 2001).

5.6.1 Smart Card File Systems

Most smart card operating systems support a modest file system based on the ISO 7816 smart card standard. Because a smart card has no peripherals, a smart card file is really just a contiguous block of smart card memory. A smart card file system is a singly rooted directory-based hierarchical file system in which files can have long alphanumeric names, short numeric names, and relative names.

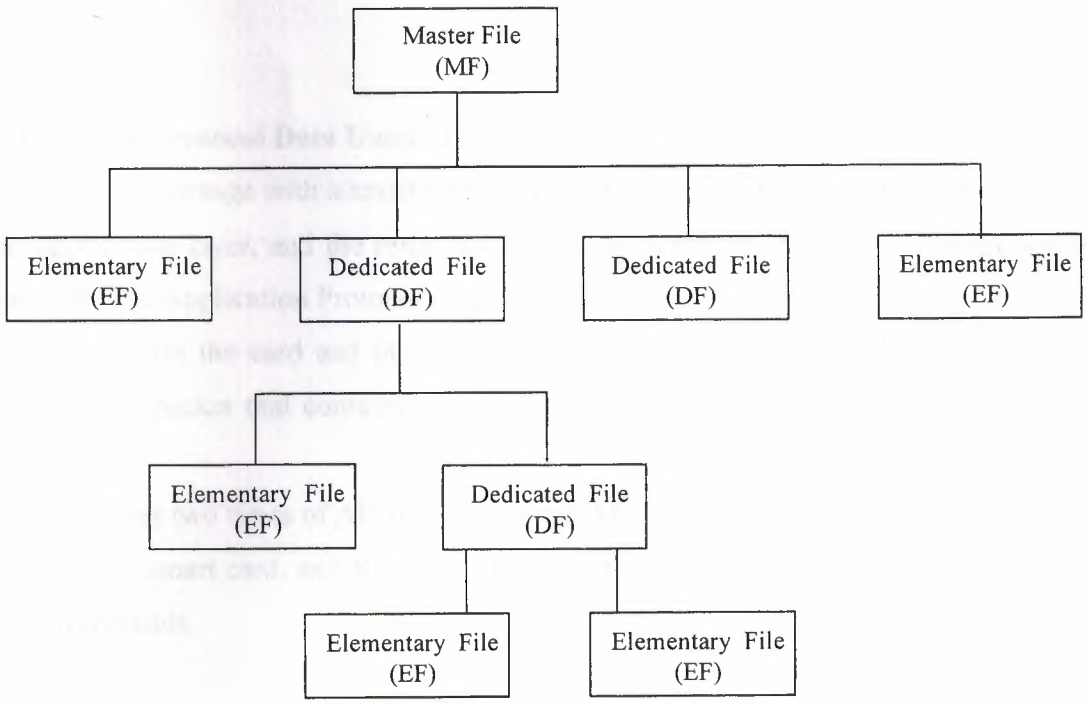


Figure 5.3 File System Architecture of Smart Card

Smart card operating systems support the usual set of file operations such as create, delete, read, write, and update on all files. In addition, operations are supported on particular kinds of files. Linear files, for example, consist of a series of fixed-size records that can be accessed by record number or read sequentially using read next and read previous operations. Furthermore, some smart card operating systems support limited forms of seek on linear files. Cyclic files are linear files that cycle back to the first record when the next record after the last record is read or written. Purse files are an example of an application-specific file type supported by some smart card operating systems. Purse files are cyclic files, each of whose records contains the log of an electronic purse transaction. Finally, transparent files are single undifferentiated blocks of smart card memory that the application program can structure any way it pleases.
(Scott Guthery & Tim Jurgensen, 1998) Associated with each file on a smart card is an access control list. This list records what operations, if any, each card identity is authorized to perform on the file. For example, identity A may be able to read a particular file but not update it, whereas identity B may be able to read, write, and even alter the access control list on the file.

5.6.2 Application Protocol Data Units (APDUs)

The basic unit of exchange with a smart card is the APDU packet. The command message sent from the application layer, and the response message returned by the card to the application layer, are called an Application Protocol Data Units (APDU).
Communication with the card and the reader is performed with APDUs. An APDU can be considered a data packet that contains a complete instruction or a complete response from a card.
ISO 7816-4 defines two types of APDUs: Command APDUs, which are sent from the off-card application to the smart card, and Response APDUs, which are sent back from the smart card to reply to commands.

APDUs consist of the following fields:

Command APDU Format

CLA	INS	P1	P2	Lc	Data	Le

Each Command APDU contains:

- A class type (CLA). It identifies the class of the instruction, for example if the instruction is ISO conformant or proprietary, or if it is using secure messaging.
- An instruction byte (INS). It determines the specific command.
- Two parameter bytes P1 and P2. These are used to pass command specific parameters to the command.
- A length byte Lc ("length command"). It specifies the length of the optional data sent to the card with this APDU.
- Optional data. It can be used to send the actual data to the card for processing.
- A length byte Le ("length expected"). It specifies the expected length of the data returned in the subsequent response APDU. If Le is 0x00, the host side expects the card to send all data available in response to this command.

Response APDU Format

Data	SW1	SW2

Each Response APDU contains:

- Optional data. 10
- Two status word bytes SW1 and SW2. They contain the status information as defined in ISO 7816-4.

The application software makes use of a protocol, which based on APDUs to exchange control and information between the reader and the card. These APDUs are exchanged by making use of the T=0 and T=1 link-layer protocols. A software component on the card interprets these APDUs and performs the specified operation; this architecture is illustrated in Figure 5.4.

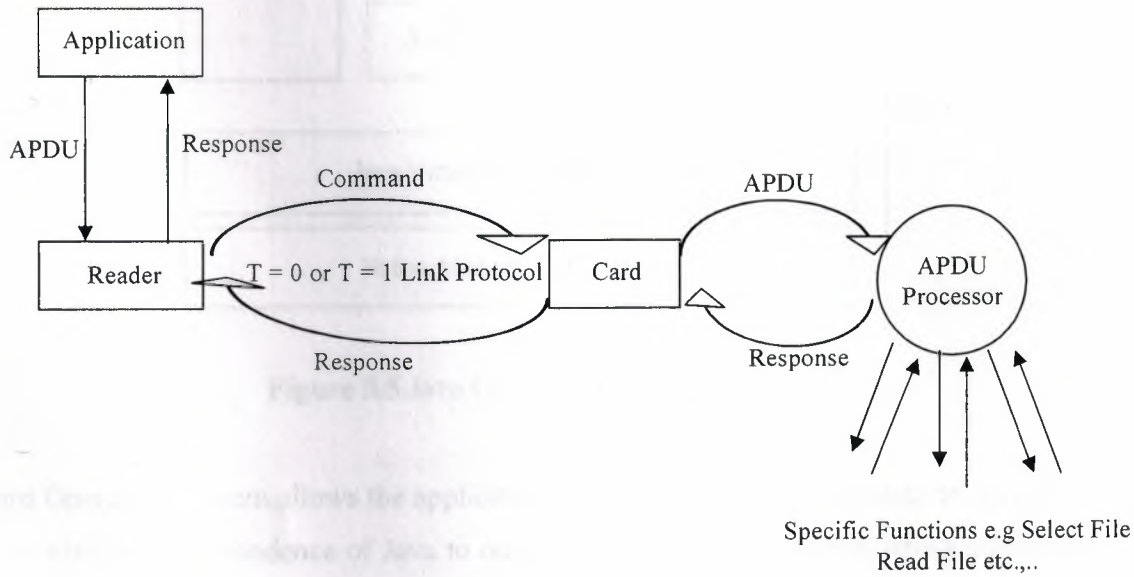


Figure 5.4 Application Communication Architecture

5. 7 JAVA Card

Java Card was introduced by Schlumberger and submitted as a standard by JavaSoft recently. Schlumberger has the only Java card on the market currently, and the company is the first JavaCard licensee. A smart card with the potential to set the overall smart card standard, JavaCard is comprised of standard classes and APIs that let Java applets run directly on a standard ISO 7816 compliant card. JavaCards enable secure and chip independent execution of different applications.

The Java Card OS Architecture is as following,

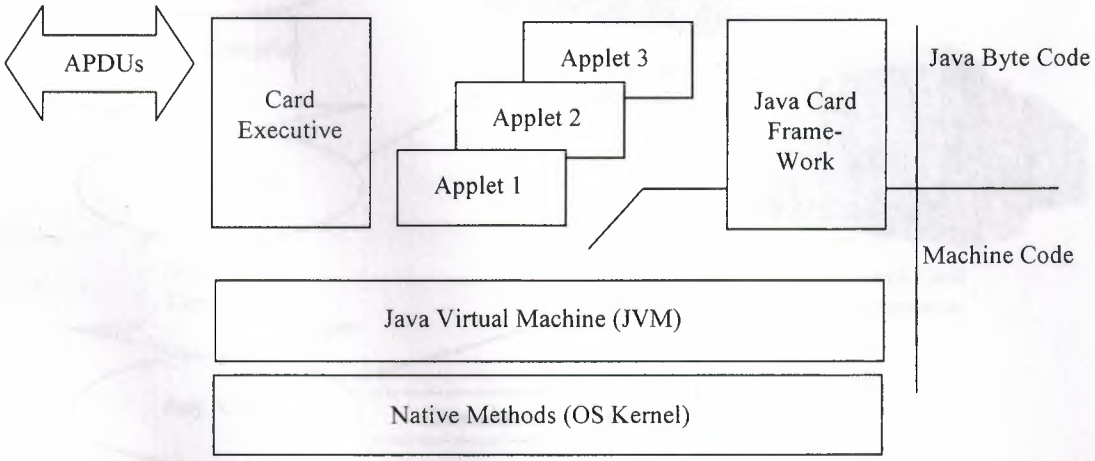


Figure 5.5 Java Card OS Architecture

Java Card Operating System allows the applications on a smart card to be written in Java. This brings the platform independence of Java to on-card software development, which used to be very proprietary for each card operating system manufacturer. In addition, it provides a good basis for multi-application cards, which support more than one application at a time. The on-card executables are referred as Card applets and consist of a Java Card Specific byte code, which is interpreted by the Java Card Runtime Environment. This runtime environment controls the execution and makes sure that different applets do not interfere.

The process for developing a Card Applet is as following,

The source files can be compiled into regular Java byte code with a standard Java compiler. The Java Card Framework is included in compiling. The "class" files could be tested in the Java Card simulation environment.

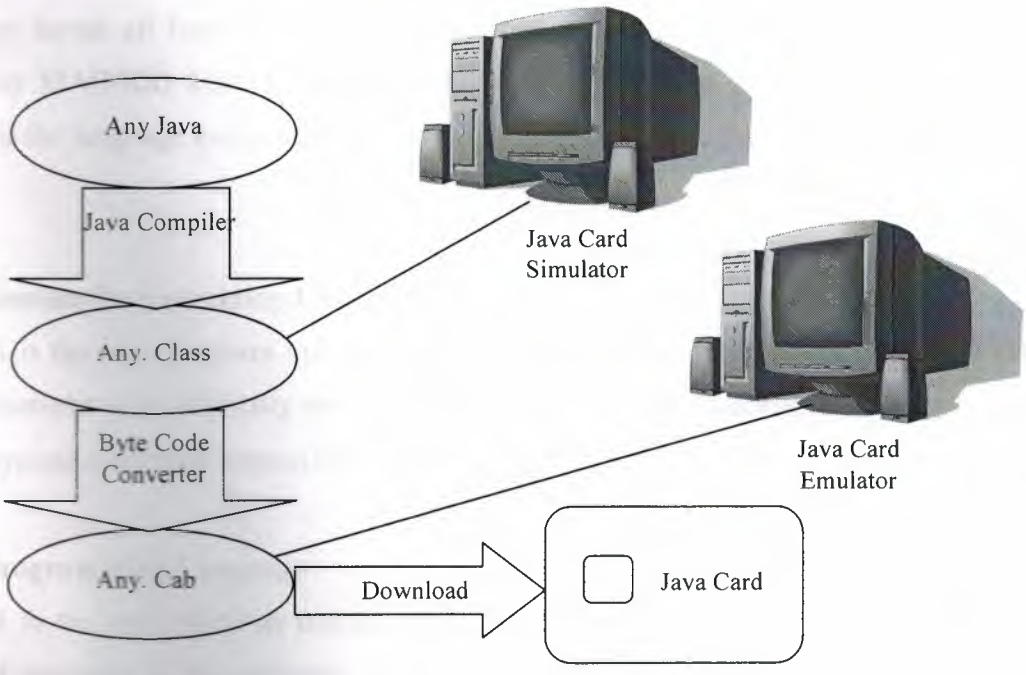


Figure 5.6 The Process for Developing a Card Applet

The byte code converter verifies the "class" file and optimizes them for the limited resources of a smart card. They are statically linked and converted into "cap" files. The "cap" files could be tested in the Java Card emulator environment.

5.8 MULTOS

MULTOS ('MULT-OS'), originally developed by Mondex International, is an open high-security multi-application operating system for smart cards, enabling a number of different applications to be held securely on the card at the same time. For application development, Mondex International has developed a smart card-optimized language: MEL (MULTOS Enabling Language), and the MULTOS-API specification. The MULTOS specification is openly licensed and controlled by leading international organizations, collectively known as the MAOSCO Consortium, that are spearheading the development and roll-out of smart card

technology across all business sectors and markets around the world. MULTOS is openly licensed by MAOSCO Ltd. (MAOSCO Consortium, 2000). The most important feature of MULOS is the language independency.

5.8.1 Assembler Programming Language

MULTOS is the only platform that has an easy to use assembler language. MULTOS smart card applications were originally developed in MEL (MULTOS Executable Language), which contains typical assembler instructions plus "smart card friendly functions" called primitives.

5.8.2 C Programming Language

MULTOS is the only platform that currently has a C compiler. This is the most common embedded language at the moment. The Swift C development tool from Swift Card is an ANSI compliant compiler that allows you to very quickly port an existing application to the MULTOS operating system. Proof of concept of most ports can be completed within two weeks.

Scott Guthery of Mobile-Mind recently ported a DOS FAT file system from GNU source code in two days having never used Swift card before. Of the Swift card C compiler Scott said "The compiler is quite rigorous and caught lots of ambiguities. The generated MEL code is very efficient. In size I found it to be only about 25% larger than 8051 native code generated by the best 8051 C compiler on the market."

5.8.3 Java Programming Language

Both Java Card and MULTOS can support the Java language. In both cases a Java compiler translates the source to Java class. For Java Card the classes are converted to Java Card byte code. For MULTOS, the Swift J compiler from Swift Card translates the Java classes (or Basic, or Modula2) to MEL code.

5.8.4 Visual Basic Programming Language

Smartcard for Windows has chosen this as their application development language of choice. To make MULTOS accessible to the VB community Swift Card Technology are currently working on a Visual Basic to MEL translator.

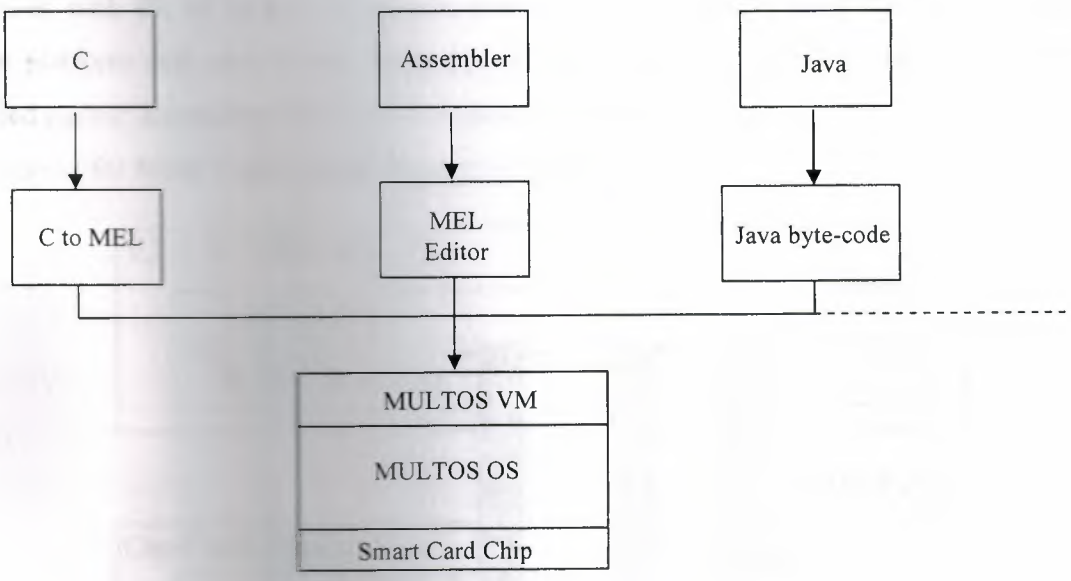


Figure 5.7 MULTOS Multi-Application Architecture

According to MULTOS materials, MULTOS is far more than an operating system. MULTOS is a complete scheme for managing smart card applications. The scheme defines a secure, efficient and cost-effective process for application management in a new world where a single card may house many applications from different sources.

MULTOS is designed and independently evaluated to a high level of security to ensure that issuers, application developers and other MULTOS service providers can build their business proposition without having to undertake expensive and lengthy evaluations of the underlying technology.

5.9 Windows Card

In 1999, Microsoft has entered the smart card Business and presented their Windows for Smart Cards operating system. As the newest member of the Windows operating system family, Windows for Smart Cards extends the benefits of the Windows environment to the smart card segment.

The Microsoft Windows for Smart Cards is an 8-bit, multi-application operating system for smart cards with 8K of ROM. (Microsoft, 2000). It is designed to be a low-cost, easy-to-program platform that runs Visual Basic applications, and is designed to meet the criteria mentioned earlier: Extending the PC environment into smart card use.

The Windows for Smart Cards Architecture is as following,

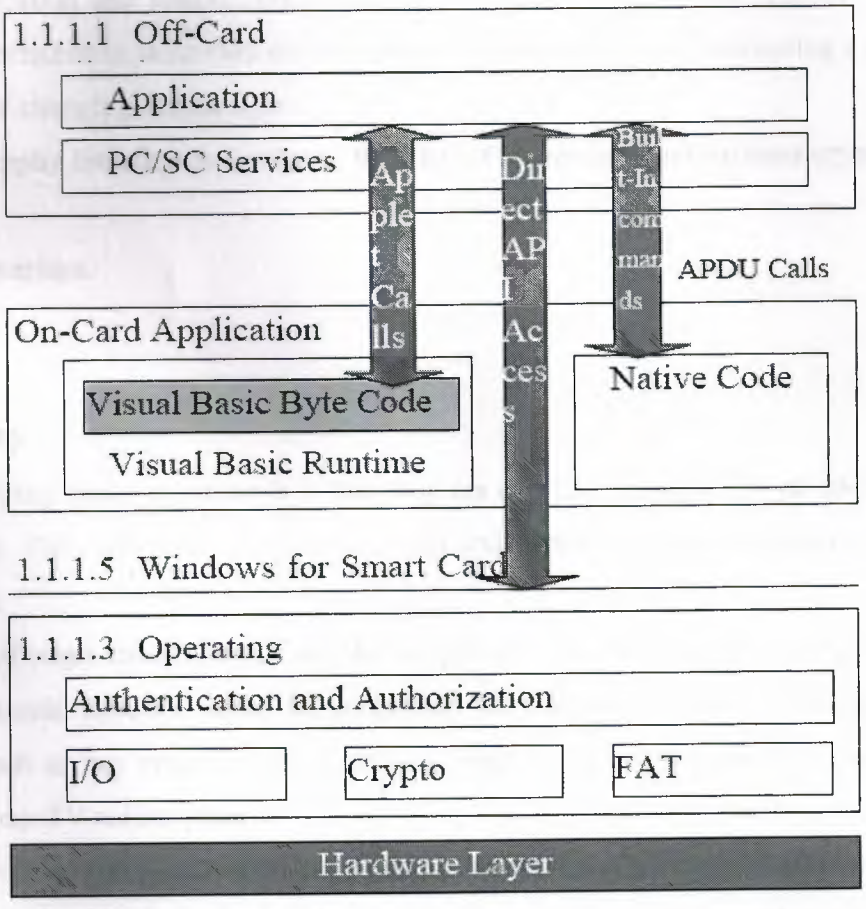


Figure 5.8 Windows for Smart Card Architecture

Quite similar to Java Card, the development of applications running on a smart card is leveraged by providing a high-level programming language. Instead of Java, Microsoft chose to use byte code generated from Visual Basic to be executed in a runtime environment on the card. The on-card applet communicates with a corresponding off-card application using common APDUs. The operating system exposes an API for working with the smart card's contents. That API is designed language-neutral and can be accessed either by Visual Basic or by native applets. It provides the following categories of functions: (Hansmann, Merk, Nicklous, Stober, 2001)

1. The File Interface includes 18 functions, for working with files.
2. The Authentication and Authorization Interface takes advantage of the Known Principals and ACL mechanisms, described earlier. There are functions for authenticating a principal, as well as for changing access rights.
3. The Cryptography Interface is similar to the PKCS#11 standard and exposes cryptographic algorithms.
4. The Utility Interface.

5.10 Summary

The important thing about smart cards is that they are everyday objects that people can carry in their pockets. They have the capacity to retain and protect critical information stored in electronic form.

The smartness of smart cards comes from the integrated circuit embedded in the plastic card. The same electronic function could be performed by embedding similar circuits in other applications, such as key rings, watches, glasses, rings or earrings. Smart keys are already being used for pay-TV subscriptions.

Smart cards are a relatively new technology that already affects the everyday lives of millions of people. This is just the beginning and will ultimately influence the way that we shop, see the doctor, use the telephone and enjoy leisure.

CONCLUSION

The important thing about smart cards is that they are everyday objects that people can carry in their pockets. They have the capacity to retain and protect critical information stored in electronic form.

The smartness of smart cards comes from the integrated circuit embedded in the plastic card. The same electronic function could be performed by embedding similar circuits in other applications, such as key rings, watches, glasses, rings or earrings. Smart keys are already being used for pay-TV subscriptions.

Smart cards are a relatively new technology that already affects the everyday lives of millions of people. This is just the beginning and will ultimately influence the way that we shop, see the doctor, use the telephone and enjoy leisure.

REFERENCES

[1]. JCI Smart Card System Consulting, 2001. Smart Card Operating System.

[2]. Sun Microsystems Inc, 2001. Java Card Technology.

[3]. <http://java.sun.com/products/javacard/>

[4]. Java Card Forum. <http://www.javacardforum.org>

[5]. MAOSCO Consortium, 2000. MULTOS: The Multi-Application Operating System for Smart Cards. http://www.multos.com/ Microsoft, 2000. Windows for Smart Card.

[6]. http://www.microsoft.com/windowsce/smartcard/ Hansmann, U., Merk, L., Nicklous, M.S., Stober, T., 2001. Pervasive Computing Handbook, 409p.

[7]. R. Merckling, A. Anderson, March 1994. RFC 57.0. Smart Card Introduction.

[8]. Scott Guthery, Tim Jurgensen, 1998. Smart Card Developer's Kit. Macmillan Computer Publishing.

[9]. Smart Card Industry Association, 2000. Knowledge Base.

[10]. Rinaldo Di Giorgio, 1997. Smart cards: A primer. Develop on the Java platform of the future. *Java World*. December 1997.